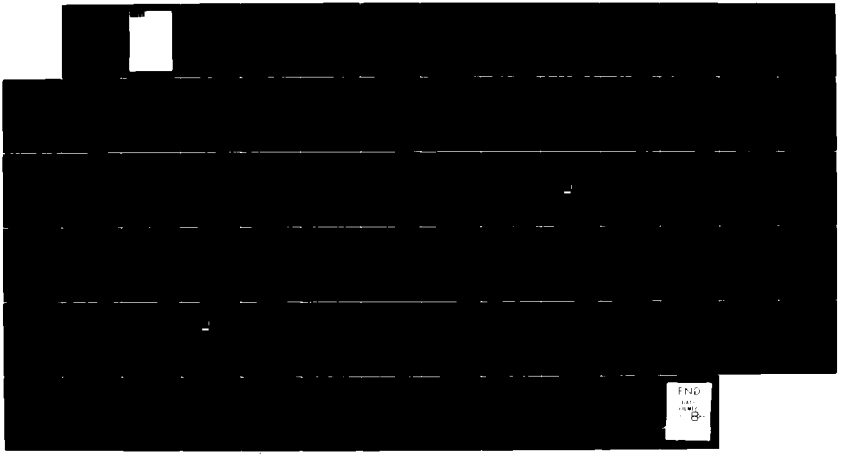
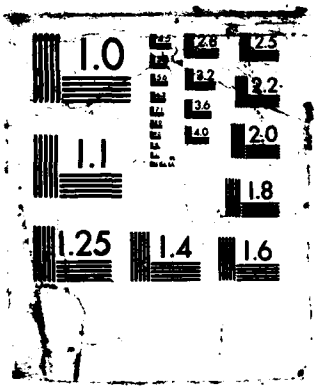
AD-A192 484 USER-EXTENSIBLE GRAPHICS USING ABSTRACT STRUCTURE(U) 1/1
ROYAL SIGNALS AND RADAR ESTABLISHMENT MALVERN (ENGLAND)
P W CORE AUG 87 RSRE-M-87011 DRIC-BR104469

UNCLASSIFIED F/G 12/5 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Report No. 87011

Title: User-extensible graphics using abstract structure

Author: P. W. Core

Date: August 1987

Summary

A means of creating an editor which allows its users to extend the classes of objects manipulated by it is described. This has been achieved by creating an abstract structure representing object classes. An example of such an editor has been implemented on Perq Flex making use of true procedure values.

CONTENTS

§ 1 Introduction

This paper is concerned with the definition and implementation of documents on computers. Documents are objects consisting of data together with a visual image of that data, displayable on a screen. Documents can be stored on backing store, edited or read by programs.

The objects, from which documents are constructed, can be divided into classes. For example, there is the class of lines of text, and there is the class of paragraphs. Most implementations of documents have the ability to include, as part of the document, objects from a predetermined set of object classes. Whilst documents are designed to include only a predetermined set of object classes, there will always be objects that users would like to include in documents which are not included in the implementation. To overcome this problem this paper provides an implementation of documents which allows users to define their own object classes and include them within the document.

Once such a document has been designed it is possible to start with a small set of object classes included in the implementation; other classes can be added by means of user-defined objects. The entire contents of a document can be thought of as an object, which may be constructed from smaller objects. It is possible to design a document which implements only user-defined classes, with the entire contents of the document being expressed as a user-defined object. Some object classes are so commonly used they can be included in the implementation of the document. A document which does not contain any user-defined objects is called a basic document.

Basic documents have been implemented on the Flex operating system on the ICL Perq [Currie, Foster and Edwards 1983, 1985]. The details of this are given in sections 3.

There are operations that are applied to documents. For example, documents can be displayed on screens, edited and coded for storage on hard disc or floppy disc. If a document containing a user-defined object is to be displayed on a screen, then there must be associated with the user-defined object a means of displaying it on the screen. Similarly, if a user-defined object is to be edited then there needs to be a means of editing it. For every operation that can be applied to a document there is a corresponding operation that can be applied to user-defined objects. This defines the objects that can be included in documents.

This paper defines an abstract structure for object classes which are to be included in documents. Any object class which is an example of this abstract structure can be included in the basic document. The structure consists of a class of data, and a set of operations to apply to the data class.

The Flex implementation of the basic document is written in Algol68. Graphical objects in the document must be specified in terms of Algol68. To accommodate this restriction the abstract structure of the graphical block is modelled in Algol68. The data of the object class are represented by values of an Algol68 type and the operations on the object class are represented by Algol68 procedures applicable to the Algol68 type of the data class.

The Algol68 model of the abstract structure is discussed in more detail in section 4.

The Flex implementation of documents provides a means of defining an object class and including objects of that class in documents. This is described in sections 5 and 6.

Section 8 gives a detailed description of the operations that can be applied to a class of objects included in a document, and this description is illustrated with an example, given in section 7. The more complex example of a simple structure editor is given in section 10.

An important aspect of the graphical objects discussed in this paper is their ability to be defined recursively in terms of other graphical objects. Section 9 gives details of how to construct graphical objects from other graphical objects.

§ 2 Notation and the use of Flex

Much of the description of the abstract structure of objects included in documents is expressed in Algol68. Where Algol68 expressions occur in this paper they are written in italics. Where sections of program are included, they are enclosed in a rectangular box.

Much of this paper is concerned with the implementation of user-defined graphics on Flex. It is intended to supply users of the Flex system with enough information to create their own graphical objects. A familiarity with Flex would aid in the understanding of those parts of the paper which are more concerned with implementation. Those parts of Flex which are used in this paper are described below.

The implementation of user_defined graphics on Flex has involved the writing of software. Some of this software is the actual implementation and the remainder is a set of tools for users creating pictures. The tools are described throughout this paper where they are relevant and the appendix includes a list of the tools together with a brief explanation of their use.

The tools are accessed via Flex Modules. Modules are objects that keep values, in this case tools, that can be used in program. The values kept by a module are accessed by including the *module* in a program and referring to the kept values by name. [Stanley, 1985] gblock_modes :Module is the notation used to reference the system module of name *gblock_modes*.

The result of compilation on Flex is the Compiledpair. This is a value which includes the code generated from the compilation, a language dependent specification of the code and the source text of the compilation. Compiledpairs are used to define user-defined graphical objects.

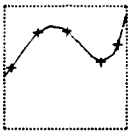
Flex has a typed operating system and a command line interpreter, curt [I. F. Currie and J. M. Foster, 1982], which checks the types of values before obeying the command line. Every value handled by the operating system has a type called a Curt Mode.

§ 3 The basic document on Flex

Objects that are displayed on documents are called graphical blocks and objects that are user-defined are called pictures. A class of objects is either a class of graphical blocks or a picture class.

Every graphical block owns a rectangle on the document and its visual image must lie inside the rectangle.

The graph occupies the area inside the rectangle defined by the dashed lines. The dotted lines would not appear on the document, but are drawn to show the limits of the graphical block.



This is a graphical block consisting of a line of characters

There are three methods of constructing graphical blocks. Firstly, there are primitive graphical object classes which form part of the basic editor; secondly, all pictures are graphical blocks and finally graphical blocks can be composed from other graphical blocks. The ability to compose graphical blocks from graphical blocks introduces recursion into the basic document.

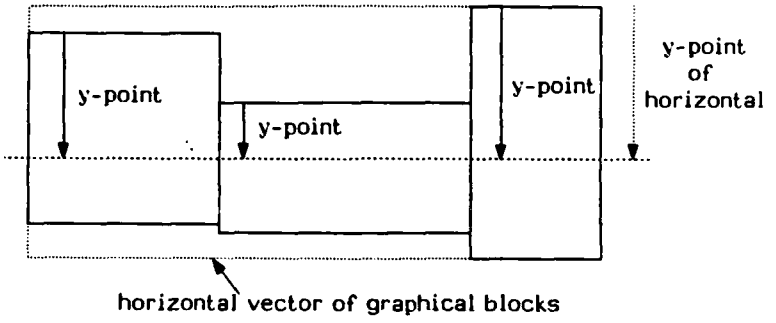
The primitive classes of graphical block include the class of lines of text and the class of black and white rectangles. Pictures are the main subject of this paper. The composition of graphical blocks from other graphical blocks is important to the subject of pictures for two reasons. Firstly, a picture may form part of a composition of graphical blocks to form a larger graphical block and secondly, pictures may be recursively defined in terms of other graphical blocks.

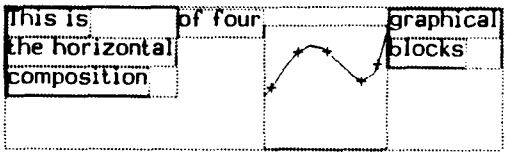
In basic documents there are two main methods of composition of graphical blocks. Firstly graphical blocks can be composed vertically. A vertical composition is formed by placing the graphical blocks vertically above each other with a left justification.

This is the vertical composition
of three
graphical blocks

The rectangle occupied by the graphical block of the vertical composition of the graphical blocks is the smallest rectangle including the composition.

Secondly graphical blocks can be composed horizontally. It is not always desirable for graphical blocks to be justified towards the top. To determine where a graphical block lies in its horizontal composition, there is a distance associated with each graphical block called its y-point. The y-point is measured from the top of the graphical block and when graphical blocks are composed horizontally, their y-points must lie on the same horizontal line.





Again the rectangle occupied by the horizontal composition of the graphical blocks is the smallest rectangle including the composition. Note that the top of the graph does not coincide with the top of the graphical block, this is because it has a different y-point. This example is more complex, with vertical graphical blocks forming part of the horizontal graphical block, illustrating the recursive nature of graphical blocks.

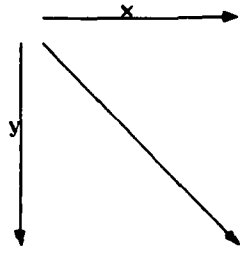
A document is constructed from a graphical block and a size. The size is the size of the document and it must not be smaller than the graphical block from which it is constructed. When the document is displayed, the graphical block is displayed in the top left of the document.

§ 3.1 GBLOCKs in programs and their composition

The basic document is written in Algol68 and this requires graphical blocks to be representable by a value with an Algol68 mode. The Algol68 mode for a graphical block is defined in the module gblock_modes :Module and is kept with the name *GBLOCK*.

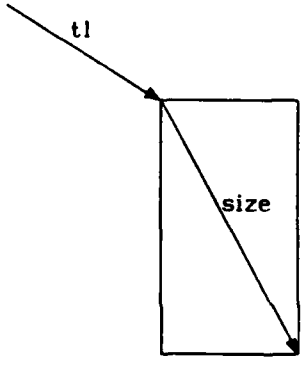
Graphical blocks own rectangles on a document, making it necessary to be able to describe vectors and rectangles in Algol68. The Algol68 modes for a vector and for a rectangle are defined in the module coord_and_re :Module and are kept with the names *VECI* and *RECT*.

VECI is a *STRUCT(INT x,y)* and represents the two dimensional vector.



Notice that the y direction is measured downwards.

RECT is a *STRUCT(VECI tl, size)* and represents the rectangle



If a graphical block is to be composed with other graphical blocks on a document, then it will be necessary to know some properties of that block. Its size will have to be known to determine where other graphical blocks lie in relation to it, and also its y-point is required

when determining how to line up horizontal vectors of graphical blocks.

The program value representing a graphical block includes both its size and its y-point.

The Algol68 mode of *GBLOCK* is

STRUCT(GITEM data, VECI size, INT ypt)

The *data* field is a union of the modes representing data classes of graphical blocks. It includes all the data classes required by graphical blocks of the basic documents and the mode *REF PICTURE*. *REF PICTURE* is the class of user-defined graphical blocks, including their data together with the operations that can be performed on that data.

§ 4 The Algol68 model of the graphical abstract structure

Every picture class must be an example of the abstract structure of objects in documents. Picture classes are defined in terms of Algol68 and there is a model for the abstract structure expressed in terms of Algol68.

The abstract structure consists of a data class and a set of operations operating on elements of the data class. The data class is defined by an Algol68 mode and can be any Algol68 mode. For example, the class of lines of text could be represented by the Algol68 mode *VECTOR[]CHAR*. The operations of the abstract structure are defined by Algol68 procedures. The modes of these procedures are entirely determined by the mode of the data class. This Algol68 model is made possible because Flex supports true procedure values [Stanley 1986].

The Algol68 model for the abstract structure is expressed in terms of a formal mode, *PICVAL*, and a set of named procedures whose modes are expressed in terms of the formal mode *PICVAL*. For example, the operation to store a graphical object on disc has mode

PROC(PICVAL)DISCCAPABILITY

Any example of this abstract structure must have a set of named procedures whose modes can be created by a substitution for the mode *PICVAL* in the abstract structure. In the case of lines of text, the operation to store the text on disc is

PROC(VECTOR[]CHAR)DISCCAPABILITY

The names and mode form of the operations follow, the name is on the left and the mode form on the right. All of the modes except for *PICVAL* are defined by the Flex implementation and are kept in common modules, *PICVAL* is the formal mode of the data class of the picture class being defined. A more detailed explanation of the specification of the operations is given in section 8.

displayer	*PROC(DISPLAYFRAME,INT,PICVAL)VOID* {displays the data}
editor	*PROC(PICTUREFRAME,INT,PICVAL)EDITRES* {edits the data}
ed_out	*PROC(PROC(CHAR)VOID,* *PROC(INT)VOID,* *PROC(GBLOCK)VOID,* *PICVAL* *)VECTOR[]CHAR* {encodes the data for output to disc or ethernet}
ed_in	*PROC(PROC CHAR,* *PROC INT,* *PROC GBLOCK* *)PICVAL* {decodes the data after reading from disc or ethernet}
val_to_disc	*PROC(PICVAL)INT* {stores the data on disc}
disc_to_val	*PROC(INT)PICVAL* {reads the data off disc}
v_split	*PROC(PICVAL,* *INT,* *PROC(PICVAL,VECI,INT)GBLOCK* *)VSRES* {splits a picture horizontally into two graphical blocks}
make_gcomp_input	*PROC(PICVAL)INPUTRES* {provides input for program}
fold_index	*PROC(PICVAL,INDEX)PICVAL* {marks a picture with an index}
to_disc	*PROC(PICVAL)GBDRES* {puts a picture to disc delivering any marks it has}

Not all of these procedures need be defined to create a picture class, but if a document contains pictures of a class, then it will not be possible to operate on those pictures with operations missing from the definition of the picture class.

§ 5 The creation of a PictureDefinition

The definition of the mode of the data class, together with the procedures representing the operations on that data, defines a class of pictures and is called a Picture Definition.

§ 5.1 The making of a PictureDefinition

The Compiledpair of the Flex operating system is the result of compilation. It contains the source text of the compilation, the code produced by the compilation and in the case of an Algol68 compilation, the Algol68 specification of the values kept by the program. The ability to hold the values resulting from compilation and their specification makes the compiledpair a suitable basis for defining a picture.

The Algol68 mode of the data class of the Picture Definition must be kept by the Compiledpair and its name must be *PICVAL*. In addition to the mode, any operations of the picture class must be kept in the Compiledpair as Algol68 procedures, in the correct form. The procedures kept must also have names so that, from examination of the Picture Definition, it is possible to determine which procedure is representing which operation. The name of each operation is the name specified in section 4.

In the Flex operating system, there is a Curt mode PictureDefinition and values of this mode have access to the defining Compiledpair of a class of pictures. To create a PictureDefinition the Flex procedure *new_picture_defn* is applied to the Compiledpair. The application of this procedure performs checks on the Compiledpair to ensure that the definition is an example of the abstarct structure. There are two checks performed; firstly the Compiledpair must keep a mode named *PICVAL*, which is then taken to be the mode of the data class of the picture definition; secondly, if any of the operations of the picture class are kept, their modes must be correct. If these two conditions are not met then the creation of a PictureDefinition will not be allowed.

§ 5.2 The definition of non-standard operations

In a Picture Definition, it is possible to define procedures that are not part of the abstract structure. These procedures are called non-standard. In order that non-standard procedures can be recognised, they must not have the name of any of the operations defining the structure.

Defining non-standard operations on pictures has to be done with care. Unlike procedures in the abstract structure, the modes of non-standard procedures are not checked.

The only reason for including non-standard operations is to enable the creator of the picture to access them from his own program which may be manipulating graphical blocks. To obtain these procedures from a picture there is a procedure *find_proc* which operates on the *PICTURE* field of the mode *GBLOCK*.

This procedure is kept in picture_proc :Module. *find_proc* has mode

PROC(PICTURE pc,VECTOR[]CHAR name)INT

This procedure cannot determine the mode of the procedure it finds and delivers a capability for the procedure described as an INT. If there is no procedure called *name* in the picture definition of *pc*, then 0 is delivered, otherwise, the mode of the result of *find_proc* will have to be changed using the binary operation 1001, defined thus.

OP(INT)PROCMODE CHANGEMODE = BIOP 1001;

It is important that *PROCMODE* is correct, otherwise the application of the procedure could result in an error.

Using the procedure *find_val*, also kept in picture_proc :Module, it is possible to obtain the data value of a picture from a graphical block. The mode of *find_val* is

PROC(PICTURE)INT

This procedure cannot determine the mode of the data of the picture and delivers a packed version of the data described as an INT. The data will have to be unpacked and given its correct mode (*PICVAL*) using the binary operation 1154, defined thus

OP(INT)PICVAL UNPACK = BIOP 1154;

This is a direct application of the Flex instruction 54.
[I. F. Currie, J. M. Foster, P. W. Edwards, 1985]

The mode *PICVAL* depends on the class of the picture from which it was obtained. It is important that the mode PICVAL is correct, otherwise errors will result.

§ 5.3 The changing of a PictureDefinition

Once a picture has been created, it may be that the operations

defining that picture need to be changed. There are two possible changes that can be made to a PictureDefinition; firstly existing procedures can be modified to incorporate changes in the specification or to correct mistakes in the definition; secondly new operations can be added to the definition. The second type of change enables pictures to be created without knowing all of the operations that may have to be performed on them.

To change a PictureDefinition a new Compiledpair has to be created which is the new definition of the class of pictures. Pictures which have already been created cannot change their data, only the operations that can operate on the data: for this reason the Algol68 mode *PICVAL* kept in the new definition must be identical to the mode in the old definition. Any of the procedures in the abstract structure kept by the new definition must be of the correct form, and if any non-standard procedure is kept in the old definition, a procedure with the same name and mode must be kept in the new definition. An arbitrary number of non-standard procedures can also be included in the new definition. When an alteration to the PictureDefinition is made, all of these properties are checked.

To make the change to the PictureDefinition the Flex procedure amend_picture_defn is applied to the PictureDefinition to deliver a procedure which operates on a Compiledpair, and will amend the PictureDefinition to the definition of the Compiledpair. This is a parallel to the Flex procedure amend used for changing the Compiledpair associated with a Module.

For example

would give the PictureDefinition the definition of the Compiledpair.

§ 6 The making of a picture from a PictureDefinition

There is a procedure on Flex (see section on Declaring Flex Picture Procedures), called make_picture_maker, which when applied to a PictureDefinition delivers a Compiledpair with an Algol68 specification. This Compiledpair keeps a procedure for making pictures (of Algol68 mode *GBLOCK*) from the class of pictures defined by the PictureDefinition. The name of this procedure is make_**name**, where **name** is the name of the Compiledpair defining the PictureDefinition.

The mode of this procedure is

PROC(PICVAL,VECI,INT)GBLOCK

This procedure is created from the Picture Definition and has the class of pictures defined by the PictureDefinition bound to it. To specify an individual picture from that class requires a data value from the data class of that picture. This is supplied as a parameter to the procedure (*PICVAL*). In addition to the data of the picture, information about the size and y-point of the picture need to be supplied.

An example of the method that may be used to create a picture on a document is

1. Make a Compiledpair keeping the mode of the data and the procedures defining the operations of the picture. Apply new_picture_defn to deliver a PictureDefinition

   Compiledpair new_picture_defn' delivering PictureDefinition

2. Apply make_picture_maker to the PictureDefinition to deliver a Compiledpair keeping make_**name**, from this make a new Module.

   PictureDefinition make_picture_maker' new' delivering
   make_**name** :Module

3. Write a program keeping a procedure to take data of a picture class create the corresponding picture and display the picture on the screen using roll.

```
roll_picture_m:
make_name :Module
groll :Module

MODE PICVAL = {mode of picture data structure};

PROC roll_picture = (PICVAL picture_value)VOID:
groll(make_name(picture_value,{size of picture},{ypt of picture}))

KEEP roll_picture

FINISH
```

§ 7 The example

The example used to illustrate the creation of a picture is the class of pictures representing polynomial fits through a set of points. The polynomial through a set of points shall be the lowest order polynomial giving an exact fit.

The data class of the pictures is the set of points through which the polynomial must pass. The Algol68 mode used to represent this data set is *REF VECTOR[]VECI*, that is a set of displacement vectors. These displacements are measured from the top left of the picture and give the position of the points.

The class of data of the picture is only a subset of the Algol68 values with mode *REF VECTOR[]VECI*. It is not possible to fit a polynomial through two points vertically above each other. A further restriction on the class of data is that the *REF VECTOR[]VECI* is ordered according to the displacement in the x direction of the *VECI*. The latter restriction is for the convenience of writing the defining operations of the picture.

The visual representation of a picture of this class is the set of points, marked by crosses, together with the line representing the polynomial fit.

The editor will have the ability to add points, remove points and move points. As the data of the picture changes, the visual representation must change to maintain consistency.

§ 8 The operations of the abstract structure of a picture

This section gives a more detailed specification of the operations in the abstract structure, illustrated by the polynomial fit example.

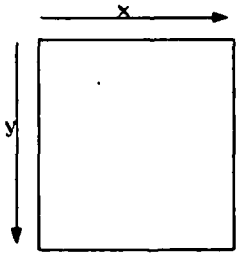
§ 8.1 The displayer

Every graphical block of a document owns a rectangle on that document, and this rectangle contains the visual image of the data of that block. The displayer is the name of the operation for displaying the visual image of the data of a picture in its rectangle.

This operation is used by the Flex editor when displaying documents on the screen and can also be used for producing hard copies of documents on printers.

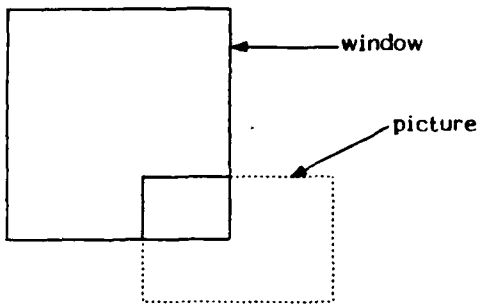
The displayer operates by assigning the visual image to a two dimensional boolean array (a raster). This boolean array is supplied via the operands of the operation and in the case of the editor it will be part of the screen. The boolean array represents pixels on the document; if an element of the array is set to *TRUE*, then the corresponding pixel is black, if it is set to *FALSE*, then the corresponding pixel will be white.

The top left pixel of the rectangle of the picture is addressed by [0,0]. The x direction is measured from left to right and the y direction is measured from top to bottom.



The element of the boolean array corresponding to (x,y) is addressed by *[y,x]*.

It is not always the case that the entire picture has to be displayed. For example, if the picture is too large to fit on the screen, or only part of the picture lies in a window on the screen, there is no need to display the picture outside that window, especially as this involves holding a larger boolean array than required.



If a document is being scrolled behind a window, then only that part of the picture which appears in the window as a result of the scroll needs to be displayed.

To stop the unnecessary displaying of parts of a picture, in addition to operating on the data of a picture, the displayer requires an additional operand with Algol68 mode *DISPLAYFRAME* (kept in pic_operator :Module). This value holds information about where to display the picture and which part of the picture to display. To obtain the boolean array to which the visual image of a picture is assigned there is an operator *AREA*, (also kept in pic_operator :Module) operating on a *DISPLAYFRAME*, delivering the boolean array. If only part of the picture is to be displayed, the boolean array is trimmed accordingly.

Pictures may have to be displayed at different magnifications, for example, the Perq screen has 100 pixels per inch, but a laser printer may have more pixels to the inch. If a document is to have the same size of visual image on both the Perq screen and the laser printer then the displaying of the picture on the laser printer must be magnified. This requirement results in a further operand to the displayer; this operand is an integer parameter and is the magnification of the visual image. A magnification of 1 corresponds to 100 pixels per inch, corresponding to the Perq screen and a magnification of *n* corresponds to 100*n* pixels per inch. The size of a picture and its y-point are measured in pixels assuming a magnification of 1. If the magnification of the visual image is n, then an untrimmed boolean array for displaying the image will have size $n \times s$, where s is the size of the picture.

The magnification of an image is part of the definition of the picture so that the user can take advantage of the increased resolution provided by the greater magnifications.

The mode of the displayer is
PROC(DISPLAYFRAME,INT,PICVAL)VOID.

DISPLAYFRAME	is the display frame
INT	is the magnification
PICVAL	is the underlying data structure

§ 8.1.1 Writing to boolean arrays and the use of display procedures

The display frame defines a boolean array to which part of the visual image of the data of a picture is assigned. If there is an attempt to assign outside of the array then a failure will occur. It cannot be assumed that the array is large enough to display the entire picture, making it necessary to determine how large the array is and what part of the picture is to be displayed.

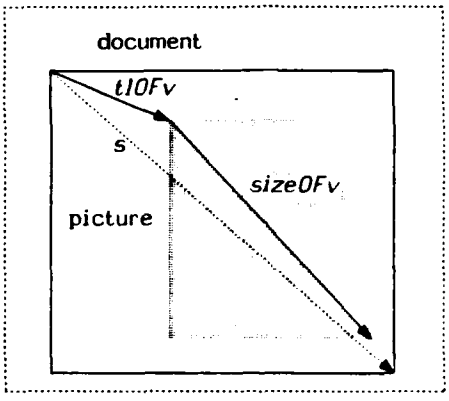
It is the responsibility of the displayer to ensure that no assignment is attempted outside the boolean array of the display frame. All of the display procedures defined in the Flex implementation and discussed in this document (see following section) are designed not to fail. They all determine the size of the boolean array and do not attempt to display outside of it.

If none of the picture to be displayed by a displayer lies in the boolean array specified by the display frame, then the call of the display procedure achieves nothing. The procedure has to determine that none of the picture is to be displayed and the checks that this involve are costly in time. If a display procedure intends to call internally another display procedure and it is easy to determine that the call of this procedure will not display any of a picture, then the speed of the displayer will be increased by not calling the internal procedure.

§ 8.1.2 DISPLAYFRAMEs and displaying procedures

In the defining of a displayer, it is often necessary to determine that part of a picture which is required to be displayed by a display frame. This can be done by applying the Algol68 operator *AREA* to the display frame and examining the bounds of the resulting boolean array, but alternatively there is an Algol68 operator *VISIBLERECT*, kept in pic_operator :Module, operating on a display frame, delivering as a *RECT*, that part of the picture which has to be displayed.

There is also an operator *SIZE*, kept in pic_operator :Module, which operates on a display frame, delivering the size of the rectangle that the picture occupies on the document.



If the shaded area is the area of the picture to be displayed, s is the vector delivered by *SIZE*, and is independent of the position of the picture on the document and the area to be displayed; *v* is the rectangle delivered by *VISIBLERECT* and is also independent of the position of the picture on the document.

There are procedures defined for drawing arbitrary lines and filling in rectangles. These procedures are defined in pic_draw_m :Module.

There are three procedures for drawing lines, *draw_horiz*, *draw_vert* and *pic_draw_line*. All three of these procedures take a pattern parameter which consists of a row of booleans. As the line is drawn, the boolean pattern is repeated. In the case of *draw_horiz* and *draw_vert* the pattern is passed as a *[]BOOL*; in the case of *pic_draw_line* the pattern is passed as a *REF VECTOR[]BOOL*. There are some predefined patterns defined in line_styles :Module. There is another parameter common to all three procedures and that is an integer *set*. *set* determines what is assigned to that part of the boolean array where the line lies.

set	Assignment to boolean array
0	The pattern.
1	The negation of the pattern.
2	The 'and' of the pattern and the boolean array.
3	The 'and' of the negated pattern and the boolean array.
4	The 'or' of the pattern and the boolean array.
5	The 'or' of the negated pattern and the boolean array.
6	The 'xor' of the pattern and the boolean array.
7	The 'xor' of the negated pattern and the boolean array.

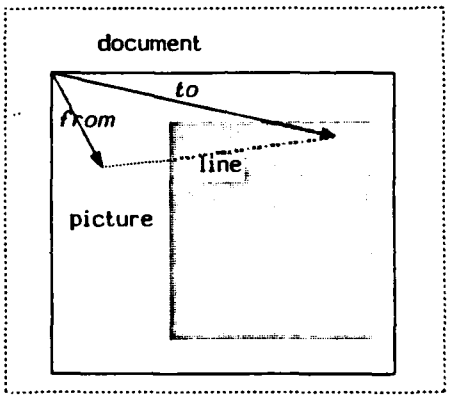
pic_draw_line is a

```
PROC(DISPLAYFRAME df,
     COORD from, to,
     REF VECTOR[]BOOL pattern,
     INT set
    )VOID
```

df is the display frame defining that part of the picture to be displayed.

from and *to* are coordinates of the positions in the picture where the line is to start and finish. *COORD* is the Algol68 mode

```
STRUCT(SHORT REAL x, y)
```



pic_draw_line does not attempt to draw outside of the shaded area even though the call of the procedure requests it.

draw_horiz and *draw_vert* draw horizontal and vertical lines, drawing them more efficiently than the more general *pic_draw_line*.

draw_horiz is a

```
PROC(DISPLAYFRAME df,
     INT y_place,
     INT from, to,
     INT set,
     []BOOL pattern,
     INT origin
    )VOID
```

df is the display frame defining the boolean array in which the line is to be drawn.

y_place is the y-position of the horizontal line.

from and *to* are the x-coordinates of the start and finish of the line.

origin is the place where the pattern starts. The line is only drawn between *from* and *to*, but the pattern is measured from *origin*.

draw_vert is a procedure with an identical Algol68 mode, with the switching of the x and y directions.

There is one procedure for filling in rectangles, this is called *fill_frame*. It has mode

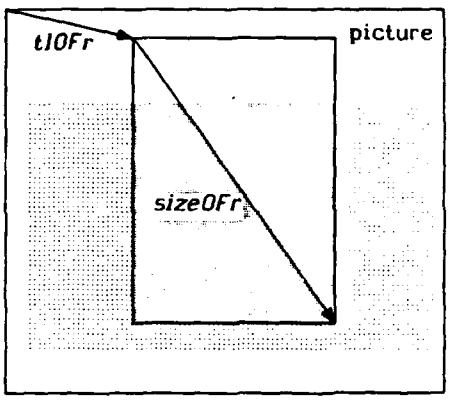
```
PROC(DISPLAYFRAME df,
     INT degree
    )VOID
```

df is the display frame defining the boolean array which is to be filled.

degree determines how the array is filled.

degree	Assignment to boolean array
1	*FALSE* (Clears the array).
2	The negation of the array.
3	*TRUE* (Blackens the array).

fill_frame fills the whole of rectangle specified by its display frame parameter to whatever degree is specified. A smaller rectangle can be filled by creating a new display frame referring to a smaller area and apllying *fill_frame* to that. There is an Algol68 operator *RESTRICT*, kept in pic_operator :Module, which operates on a display frame and a rectangle, delivering a new display frame which is the old display frame restricted to the rectangle.



If *df* is a display frame specifying the displaying of the picture in the lightly shaded region, then *df RESTRICT r* is a new display frame specifying the area that is more darkly shaded.

§ 8.1.3 The displayer of polynomial fit

The displayer of polynomial fit is divided into two sections; the first section displays crosses where the points lie, and the second section plots the curve.

```
MODE PICVAL = REF VECTOR[]VECI;

PROC displayer = (DISPLAYFRAME df, INT mag, PICVAL pv)VOID:
 (REF VECTOR[]BOOL solid=styles[1];{style of line to be drawn}
  FORALL pv_i IN pv DO {forall points}
    draw_vert(df,xOFpv_i,yOFpv_i-4,yOFpv_i+4,0,solid,0);
    draw_horiz(df,yOFpv_i,xOFpv_i-4,xOFpv_i+4,0,solid,0)
  OD;

  IF UPB pv>1 THEN draw curve FI {plot curve}
 )
```

The crosses are composed from solid horizontal and vertical lines of length 9, intersecting at the point to be marked. If there is more than one point, then the curve is plotted.

The curve plotting procedure:-

```
INT order=UPB pv;
REF [,]REAL eqns=HEAP [1:order,1:order+1]REAL;
find fit and store coeffs in eqns[,order+1];
[]REAL coeff = eqns[,order+1];

PROC f = (REAL x)REAL: {define function}
(REAL s := coeff[order];
 FOR i FROM order-1 BY -1 TO 1 DO s:= s*x+coeff[i] OD;
 s
);

PROC plot_x_int= (INT low, high)VOID:
 (draw line from low_x to high_x);

RECT r = VISIBLERECT df;
INT  left = xOFtlOFr,
     right = left+xOFsizeOFr-1;

{find first point in visible area}
INT start := 0;
FOR i TO order
WHILE  xOFpv[i]<left OREL(start:=i;FALSE)
DO SKIP OD;

IF start=1 THEN plot_x_int(0,xOFpv[1])FI;{draw first line}
IF start/=0 THEN
   {draw lines between points whilst inside visible area}
  FOR i FROM (start MAX 2) TO order WHILE
    INT last = xOFpv[i-1];
    plot_x_int(last,xOFpv[i]);
    i<=right
  DO SKIP OD
FI;
IF  xOFpv[order]<right THEN
  plot_x_int(xOFpv[order],xOF SIZE df) {draw last line}
FI
```

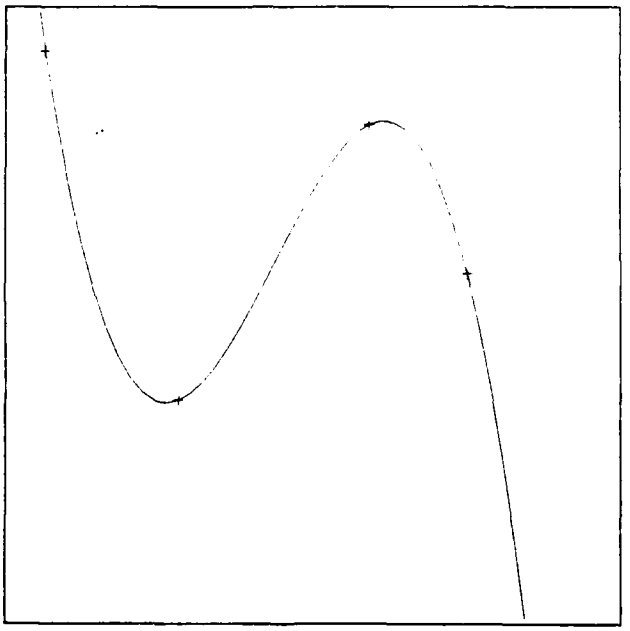
The coefficients of the polynomial that fits the points in the data of the picture are calculated and stored in *coeff*. The function *f* is the polynomial.

plot_x_int draws the curve representing *f* between the two x points supplied as a parameters. It only displays the curve where it lies in the area specified by the display frame *df*. The curve is drawn as a series of straight line approximations using *pic_draw_line*, the x displacement of each line being 10 and the first line starts at the x

position *low*. If the same part of the curve is drawn twice, and the starting point for the second drawing does not coincide with the end of a straight line in the first drawing, then the straight line approximations of the two curve drawings will not coincide. For this reason *plot_x_int* is only called on the intervals between the points of the data of the picture; that is it can be called on the interval between any two points, but not on any sub-interval

It is possible to draw the curve by the calls of *plot_x_int* on the entire interval [0, *x OF SIZE df*]; however it is only necessary to call the procedure on those inter-point intervals which lie in the area specified by the display frame.

An example display:-



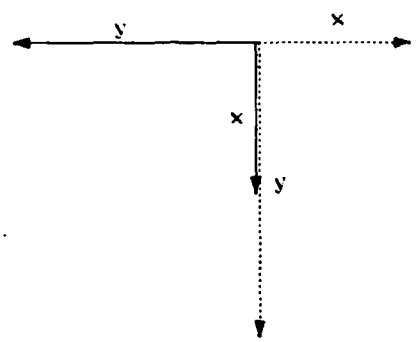
§ 8.1.4 Sideways displaying

Sometimes it is necessary to display pictures that have been rotated through 90 degrees, for example displaying documents in a landscape manner on a laser printer. It is much more efficient to write the pixels to the boolean array as they are to be displayed, rather than to write them as portrait and rotate them afterwards.

The information on whether to display a picture sideways lies with the boolean array to which the visual image is assigned. If the upper bound of the second dimension of the boolean array is negative, then

the picture must be displayed sideways. The procedures defined in the implementation, such as *pic_draw_line*, will draw their images sideways if this is specified by the boolean array in the display frame.

If users are writing directly to the boolean array where the picture is to be displayed, then they will have to take account of the rotation. A point that normally addressed by *[y,x]*, is now addressed by *[x,-y]*.



§ 8.2 The editor

The Algol68 mode of the editor of a picture is

PROC(PICTUREFRAME, INT, PICVAL)EDITRES

where

EDITRES = STRUCT(BOOL altered, INT reason, PICVAL picval)

This operation is called by the Flex operating system and may be called for one of several reasons.

The editor operates on three values; the *PICTUREFRAME* is the context of the picture on the document, the *INT* is the reason for calling the editor and the *PICVAL* is the data of the picture being edited. The editor delivers three values, a *BOOL* which is *TRUE* if the data of the picture has altered, an *INT*, which is the reason why the editor was left and a *PICVAL* which is the new data of the edited picture.

§ 8.2.1 Reasons for entering or leaving the editor

The editor of a picture is usually called by the Flex editor when it is editing a document which contains the picture.

The editor of the picture can be called for many reasons. The most usual reason is that the cursor has been placed on the picture because the user wishes to change the data of the picture. The picture editor is also called if the editor of the document, on which the picture lies, is searching for a string of characters. The picture editor would be expected to determine whether or not it contained such a string; if it contained a string, it should place the cursor on the string and wait for a request for action from the keyboard; otherwise it should exit.

When the editor of a picture is called, there is an integer passed to it as a parameter. This integer is a coding of the reason for the calling of the editor. The set of integers and corresponding Algol68 names used for coding are kept in pic_reasons_ :Module.

§ 8.2.1.1 Requests for actions and their implementation

The actions of an editor are normally determined in one of two ways. Firstly, the reason supplied to the editor on its calling may entirely determine what it has to do; for example if the reason was *find_string* and the picture did not contain the string being searched for, the editor is exited. Secondly, the editor responds to the reading of the keyboard and the puck.

The reading of the keyboard is done via a procedure delivering an integer which is a coding of the key/puck button pressed. Since most readings of the key/puck require some form of action by the editor, the integer delivered can be considered as a coding of the action to be performed by the editor.

It is not possible for the editor of a picture to perform all the actions that may be required of it. Suppose the picture was the second element in a horizontal composition of graphical blocks and the first element was a line of text. If the required action of typing the key 'c' was to put the character 'c' immediately in front of the picture, i.e append the character to the line of text, (as happens when editing a cartouche), the picture editor could not do this since it has control of the picture but not anything outside of it. The editor of the document, which calls the editor of the picture, could place the character 'c' in front of the picture. In order that the document editor can do this, the picture editor has to be left and the action required of the document editor communicated. The action required of the document editor is passed as its integer coding as part of the result of the call of the picture editor.

In addition to the actions resulting from direct interpretations of the keyboard/puck, it is possible for the editor of the picture to request

other actions of the document editor by delivering other action codings.

§ 8.2.1.2 The integer coding of reasons of entering the editor

The module pic_reasons_ :Module keeps integer codings of actions and reasons, the names given are intended to suggest the actions they cause when passed to and from the document editor. These codings appear in italics.

find_next_mark
The Flex editor is searching for the next mark in a document and has found a picture. If the data of the picture includes the ability to mark parts of the picture then the cursor should be placed where the mark is, writing the message associated with the mark on the display line and then normal editing should continue. If the picture does not contain any such marks then the picture editor should be left with the action *find_next_mark*, thereby informing the Flex editor to continue searching for a mark in the remainder of the document.

find_string
The Flex editor is searching for a string in a document and has found a picture. If the picture has such a string then the cursor should be placed on the string and normal editing continued. If the picture does not contain such a string then the picture editor should be left with the action *find_string*, thereby informing the Flex editor to continue searching for the string.

go_in
The Flex editor is trying to determine the graphical block nearest to the top left of the cursor. It has found the picture. After the select button of the puck is released, the top left of the cursor is set to the position of the puck.

move_down, move_left, move_right, move_up & southwest
The editor is moving from one graphical block on a document to another in the direction of the reason and the graphical block it has found is the picture.

§ 8.2.1.3 The integer coding of the puck/keyboard

The reading of the keyboard/puck delivers an integer coding as specified in the following list:-

If the key for a character is pressed, then its code is delivered and is in the range 32 ... 127. If **CNTRL** is pressed in conjunction with a character key, then 128 is added to the code.

Key	Reason	CNTRL Key
SET UP	*find_string*	*6*
HELP	*help*	*135*
↑	*move_up*	*28*
↓	*move_down*	*29*
←	*move_left*	*30*
→	*move_right*	*31*
OOPS	*southwest*	*149*
ACC/ESC	*double_edit*	*155*
BACK SPACE	*backspace*	*136*
BREAK	*128*	*129*
TAB	*find_next_mark*	*137*
REJ/DEL	*127*	**BREAK IN**
NO SCROL	*change_mode*	*14*
LINEFEED	*10*	*quit*
PF1	*delete_char*	*delete_char*
PF2	*duplicate*	*duplicate*
PF3	*split_page*	*split_page*
PF4	*delete*	*delete*
ENTER	*evaluate*	*evaluate*
KEYPAD .	*ins_blank_below*	*ins_blank_below*
KEYPAD 0	*container*	*container*
KEYPAD 1	*cart_to_mode*	*cart_to_mode*
KEYPAD 2	*line_to_para*	*line_to_para*
KEYPAD 3	*tidy_para*	*tidy_para*
KEYPAD 4	*split_horiz*	*split_horiz*
KEYPAD 5	*undo_cartouche*	*undo_cartouche*
KEYPAD 6	*group*	*group*
KEYPAD ,	*insert_del_horiz*	*insert_del_horiz*
KEYPAD 7	*insert_del_vert*	*insert_del_vert*
KEYPAD 8	*prev_page*	*prev_page*
KEYPAD 9	*next_page*	*next_page*
KEYPAD -	*ins_blank_above*	*ins_blank_above*
PUCK select	*track*	*track*
PUCK examine	*examine*	*examine*
PUCK result	*result*	*result*

§ 8.2.1.4 The integer coding of actions resulting from an edit

When a the editor of a picture is exited, the *reason* field of the result of the edit determines the action taken by the editor of the document.

If a number in the range 32 ... 127 is delivered, then the character corresponding to that code is inserted in front of the picture and the picture editor is re-entered with reason *go_in*.

If *225* is delivered, then the 'Control a' display appears, operating on the picture.

If the **CNTRL** codes for the letters a ... z, A, C..S, U..Z are delivered, then if these keys are set, their respective action will be applied to the picture, otherwise there will be a beep and the picture editor will be re-entered with reason *go_in*.

If the code for **CNTRL B** is delivered the editor will move to the bottom of the document.

If the code for **CNTRL T** is delivered the editor will move to the top of the document.

backspace
If there is a character immediately before the picture, it is deleted, the picture editor is re-entered with a reason of *go_in*.

come_back
The Flex editor determines the graphical block nearest to the top left of the cursor and calls its editor with a reason of *go_in*.

container
The cursor will be placed on the graphical block containing the picture.

delete
The picture is put on the list of remembered elements.

double_edit
The window containing the picture will be split and the picture editor re-entered with reason of *go_in*.

duplicate
The picture will be placed on the list of remembered elements. If there is a following graphical block in the document, the cursor will be placed on it, otherwise there will be a beep and the picture editor is re-entered with a reason of *go_in*.

evaluate
The line containing the picture is evaluated by curt.

find_next_mark
The Flex editor searches for the next mark in the remainder of document after the picture.

find_string
The Flex editor will search through the remainder of the document

after the picture, for the string of characters set up with the FINDLINE key.

group
This causes the picture to be grouped in the same manner as if it were a cartouche.

ins_blank_above & ins_blank_below
This causes a blank line to be inserted either above or below the picture.

ins_del_vert
This causes the top of the list of remembered elements to be inserted above the line containing the picture.

line_to_para
This causes the line containing the picture to be turned into a picture.

move_left, move_right, move_down, move_up & southwest
This causes the cursor to be placed on the graphical block in the corresponding direction.

next_page & prev_page
If the picture lies in a paged structure, there is a passage through the pages in the forward and backward directions respectively.

quit
This causes the editor to quit from the edit of the document

result
This causes the editor to result from the edit of the document.

split_page
This causes the page to be split immediately above the line containing the picture.

tidy_para
If the picture is in a paragraph, the paragraph will be tidied.

track
The puck is tracked and when the select button is released the top left of the cursor is set to the puck position and the editor of the graphical block nearest the puck is entered with a reason of *go_in*.

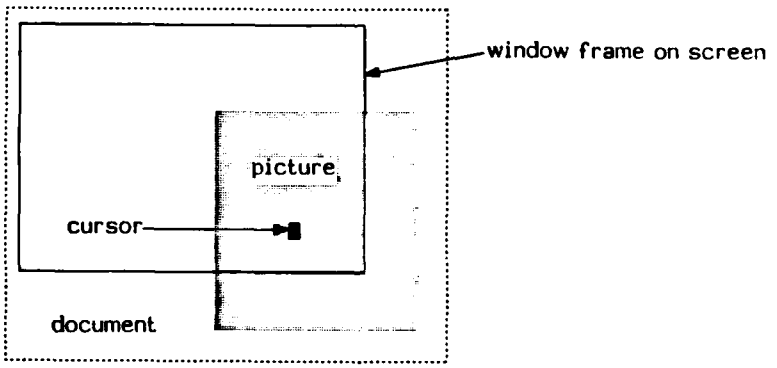
All other actions cause a beep and a re-entry of the editor with a reason of *go_in*.

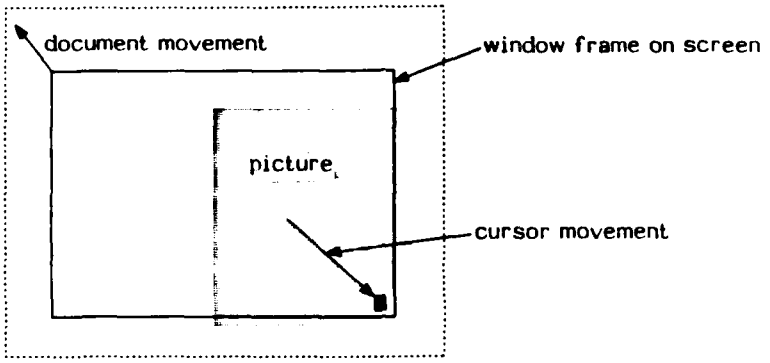
§ 8.2.2 Altering the data of the picture

The editor of a picture class essentially operates on the data of a picture together with its visual image, delivering another element of the data class of the picture together with its visual image. The visual image is usually displayed when the picture editor is called and must be kept consistent as the data of the picture is altered. In addition to the data, the editor delivers a boolean which informs the editor of the document whether or not any change to the data of the picture is to be recorded should the document containing the picture be put to disc. If, on every edit of a picture, the editor delivers a boolean signifying no alteration, then when the document is put to disc the picture shall return to the form that it had when it was taken from disc; otherwise, when the document is put to disc, the picture shall have the data of the result of the last edit. If the picture had never been stored on disc the picture resulting from the last edit is stored.

§ 8.2.3 Movement, expansion and the PictureFrame

Whilst editing a picture it may be necessary to move the document on which the picture lies behind the window through which it is seen. This happens when not all of the picture can be seen through the window and the document has to be moved to include other parts of the picture.





The movement of the document behind the window brings into view parts of the document both inside and outside of the picture. Although the picture editor can determine how to display that part of the document included in the picture, because it has the data of the picture as an operand, it requires information on how to display the area outside of the picture. This and other information is passed through to the picture editor via an operand with an Algol68 mode *PICTUREFRAME*, kept in the module pic_operator :Module. This can be thought of as an abstract data type with its operations, such as delivering how to display the cutside of a picture, provided by Algol68 procedures and operations.

There is a procedure *pic_scroll*, kept in pic_scroll_m :Module, which moves a document behind the window whilst editing a picture. The ability to display both the picture and its surrounding is required by this procedure making the procedure take three parameters: a *PICTUREFRAME*, which gives the ability to display areas outside of the picture; a *PROC(RECT)VOID*, this is the displayer for displaying an arbitrary rectangle of the picture; and a *VECI* which is the vector giving the change in the position of the document behind the window.

It is often possible to create the *PROC(RECT)VOID* from the displayer of the picture thus

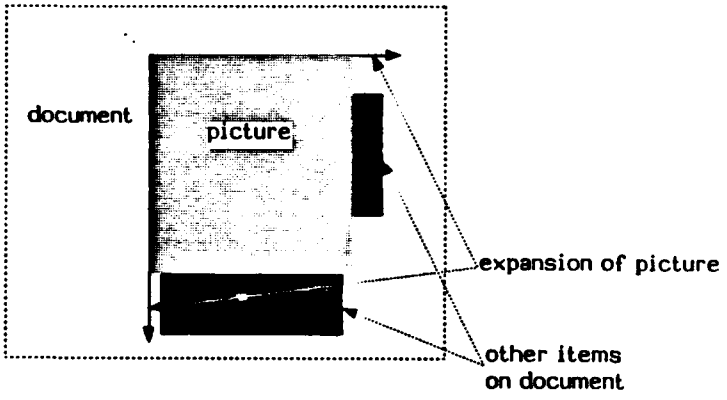
```
(RECT r)VOID:displayer(DISPFRAME pf RESTRICT r,picval,1)
```

DISPFRAME is an operator, kept in pic_operator :Module, for converting a picture frame into a display frame. When a picture is being edited, there is a part of the screen to which it assigns for displaying its visual image. The display frame for this boolean array is obtained by applying *DISPFRAME* to the picture frame of the edit. It is important to realise that the display frame does not remain

constant throughout the edit, but changes as the picture changes and moves on the screen.

The procedure for displaying an arbitrary part of the picture on the screen obtains the display frame from the picture frame, *pf*, passed to the picture editor, restricts the display frame so that it only includes the rectangle *r*, and with this display frame, displays the current value of the data of the picture, *picval*, with a magnification of 1, which is the magnification of the screen.

Whilst editing a picture a change of its data may result in a change in its size. The top left point of the picture is fixed relative to the document, so that any change of size will result in an expansion or contraction either to the right or to the bottom. This change to a picture may affect the position of other objects on the document; they may have to move to allow room for the picture to expand.



If the document in this diagram is of fixed size, then the expansion shown cannot take place since this would require the movement of the item below the picture so that part of it would lie outside of the document; this is not allowed. If the document is flexible in the y direction then the document could expand to allow the item below the picture to be moved downwards to allow for the increase in size of the picture. Whether or not the document is flexible in the x direction, there is room for the expansion of the picture in the x direction.

The ability to expand a picture depends on the context of the picture and the context of a picture is held in the picture frame. To expand or contract it is necessary to apply the Algol68 operator *EXPAND*, kept in pic_operator :Module, to the picture frame of the picture. *EXPAND* operates on a *PICTUREFRAME* and a *VECI*, the *VECI* being the

amount of expansion. *EXPAND* delivers a boolean which is *TRUE* if the expansion is allowed and *FALSE* if it is not. Expansions are effected by expanding with a positive amount and contractions by expanding with a negative amount. If there is an expansion in one direction there must not be a contraction in the other.

When *EXPAND* is called, it may have to change the location of other objects on the document to accommodate the change of the picture. When expanding, if there is room for the expansion, *EXPAND* clears part of the document to give space for the expansion of the picture. When contracting, the editor of the picture must clear that part of the picture from which it is contracting before calling *EXPAND*, thereby leaving space for the other objects on the document to move into during the call of *EXPAND*. A call of *EXPAND* with a contraction is always allowed.

§ 8.2.4 Reading the keyboard and the puck

The state of the puck can be read at any time during an edit and is read by the procedure *read_puck*, kept in read_puck_m :Module. *read_puck* has mode *PROC(PICTUREFRAME)PUCKRES* where *PUCKRES* is a *STRUCT(VECI posn, BITS buttons)*. The parameter is the picture frame of the edit; *posn* is the position of the puck relative to the top left of the picture and *buttons* are the buttons that are held down on the puck. When the select button is depressed, the least significant bit is set; when the examine button is depressed, the second bit is set and when the result button is depressed, the third bit is set.

There is a procedure *inner_read*, kept in inner_read_m :Module, which performs two functions. It moves the document so that it includes the cursor and displays the cursor; it then waits until a key is pressed, either on the keyboard or on the puck. When the key is pressed the integer coding for that key is delivered. Since *inner_read* can cause a movement of the document behind the window, it will require the ability to display arbitrary parts of the document. The picture frame provides the ability to display all that is outside the picture and a *PROC(RECT)VOID*, displaying an arbitrary rectangle in the picture, is also required. The mode of *inner_read* is

PROC(PICTUREFRAME, PROC(RECT)VOID)INT

The cursor is removed from the document when *inner_read* is exited.

There is another procedure *read*, kept in read_m :Module, which is almost identical to *read* except that it waits for the examine and result buttons on the puck to be released before delivering their integer coding, instead of delivering the coding the moment when they are pressed.

§ 8.2.5 Control of the cursor

The cursor is a rectangle within the picture and is displayed in the call of either of the two procedures *inner_read* or *read*.

The current location of the cursor can be read using the procedure *give_cursor_rect*, kept in cursor_contr :Module. *give_cursor_rect* requires the picture frame as parameter and delivers the rectangle of the cursor relative to the top left of the picture.

To set the location of the cursor, there is a procedure *set_cursor_rect*, also kept in cursor_contr :Module, which requires a picture frame and the rectangle to which the cursor is to be set : it sets the cursor to that rectangle.

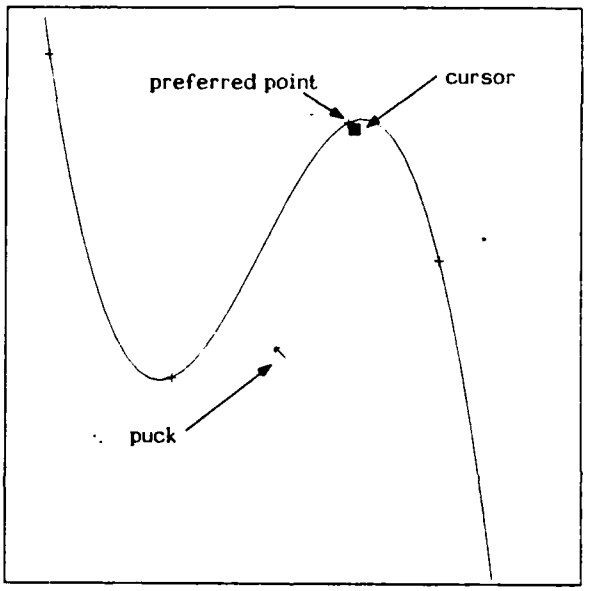
§ 8.2.6 Following the puck

An important part of the editor is the handling of the action *track*. *track* is delivered by a call of *inner_read* or *read* when the select button is pressed. The normal conventions of the editor are that the puck is followed whilst the select button is pressed and when it is released the cursor is placed on the nearest element to the puck. This may involve scrolling the document behind the window.

The simplest method of achieving this is to deliver from the picture editor the reason *track*. The tracking is taken care of by the procedure calling the picture editor which is usually the editor of the document. The calling procedure can then scroll the screen if necessary and when the select button of the puck is released the calling procedure can set the top left of the cursor to the position of the puck and return with a reason of *go_in*. In the calling editor the puck is tracked and, when the puck is released, the editor of the graphical block nearest to the puck is called with a reason of *go_in*. This is not necessarily the editor of the picture which was exited to start the tracking of the puck.

§ 8.2.7 The polynomial fit example

The editor of polynomial fit provides only three operations. It accepts new points, deletes existing points and moves existing points. There is always a preferred point (unless there are 0 points) and this is initially the point nearest to the puck (the point whose x displacement is closest to the x displacement of the puck) when the select button is released. The preferred point is marked by a cursor of size (10,10). When the editor moves or deletes a point, it is the preferred point that is moved or deleted. When a point is moved, it is moved to the current position of the puck and remains the preferred point. When a new point is inserted, it is inserted at the current puck position and it becomes the preferred point.


preferred point
cursor
puck

The editor of polynomial fit:-

```
PROC editor = (PICTUREFRAME pf, INT reason, PICVAL pv)EDITRES:
 IF reason=find_string OREL reason=find_next_mark THEN
  (FALSE,reason,pv)
 ELSE
  REF VECTOR[]VECI npv := pv;
  BOOL altered:=FALSE;
  INT action,preferred_point;

  PROC find_preferred_point = VOID:find_preferred_point;
  PROC add_point =(VECI point)VOID:add_point;
  PROC remove_point =(INT place)VOID:remove_point;
  PROC loc_disp=(RECT r)VOID:
    displayer(DISPFRAME pf RESTRICT r,1,npv);
  PROC redisplay = VOID:
   (RECT pr = VISIBLERECT pf;
    pf CLEAR pr; loc_disp(pr)
   );
  find_preferred_point;
  WHILE
   action:=inner_read(pf,loc_disp);

   IF action = ins_del_vert THEN
    add_point(posnOFread_puck(pf));
    redisplay;TRUE
   ELIF action = delete THEN
    IF UPB npv<1THEN warning("No points to delete")
    ELSE
     remove_point(preferred_point);
     find_preferred_point;
     redisplay
    FI;
    TRUE
   ELIF action = ABS "m" THEN
    IF UPB npv<1THEN warning("No point to move")
    ELSE
     remove_point(preferred_point);
     add_point(posnOFread_puck(pf));
     redisplay
    FI;
    TRUE
   ELSE
    FALSE
   FI
  DO SKIP OD;
  (altered,action,npv)
 FI
```

The picture does not contain any strings or marks so that if the

editor is entered looking for either of these it is left immediately with the same reason, thereby informing the calling procedure to continue looking. The data structure has not been altered so it is returned together with a boolean value false.

Any other reason of entry to the editor causes entry into its main loop. Once in the main loop, the editor reads the puck/keyboard to determine what action the user wishes. If the action is one of the three that the editor can perform, it completes the action and goes around the loop again. All other actions are communicated to the calling procedure by exiting with the action as a result. As a result of obeying actions the data of the picture changes and the current value of the data is stored in *npv*, the recording of an alteration is stored in *altered*. The preferred point can also change as a result of actions and is stored as an index into the vector of points in the variable *preferred_point*.

There are five sub procedures of the editor:-

1. *find_preferred_point* reads the top left of the current cursor position and assigns to *preferred_point* the index of the point nearest to it. It also moves the cursor to the preferred point and sets its size to (10,10). If there are no points in the data structure, then the cursor's size is set to (0,0) which makes it invisible. If the editor is entered by releasing the puck then the top left of the cursor holds the current position of the puck and consequently *find_preferred_point* finds the nearest point to the puck.

```
PROC find_preferred_point = VOID:
IF UPB npv = 0 THEN set_cursor_rect(pf,((0,0),(0,0)))
ELSE
  INT x = xOFtlOFgive_cursor_rect(pf);{x-coordinate of cursor}

  {find  lowest point larger than cursor}
  INT place:=UPB npv;
  FOR i TO UPB npv WHILE xOFnpv[i]<x OREL(place:=i;FALSE)DO
    SKIP
  OD;

  {If cursor is nearer lower one then decrement place}
  IF place>1ANDTH (xOFnpv[place]-x)>(x-xOFnpv[place-1])THEN
    place-:=1
  FI;

  {set the cursor to the place}
  set_cursor_rect(pf,(npv[place],(10,10)));

  {make place the preferred point}
  preferred_point := place
FI
```

2. *add_point* adds a new point into the data structure. When adding a new point it is important to preserve the property that all the points in the vector of points are ordered according to their x-coordinate. Whenever a new point is added, it becomes the preferred point so that the cursor is set to this point and its index in the vector is assigned to *preferred_point*. It records the alteration to the data structure by setting *altered* to *TRUE*.

```
PROC add_point =(VECI point)VOID:
(VECI size = SIZE pf;

 {make the point lie in the picture}
 VECI pt=((xOFpoint MIN xOFsize) MAX 0,
          (yOFpoint MIN yOFsize) MAX 0);

 {make new vector 1 larger than old}
 INT place:=UPB npv+1;
 HEAP VECTOR[place]VECI temp;
 altered := TRUE;  {record alteration}

 {find place to insert new point}
 FOR i TO UPB npv
 WHILE xOFpt>xOF npv[i] OREL (place:=i; FALSE)
 DO
  SKIP
 OD;

 {insert new point}
 temp[:place-1]:= npv[:place-1];
 temp[place]:=pt;
 temp[place+1:]:=npv[place:];
 npv:=temp;

 {set the cursor to new preferred point}
 set_cursor_rect(pf,(pt,(10,10)));

 preferred_point := place
        {make the new point the preferred point}
)
```

3. *remove_point* takes an integer parameter, which is the index of the point in the vector, removes the point from the vector of points and records the change in the data structure by setting *altered* to *TRUE*.

```
PROC remove_point =(INT place)VOID:
(HEAP VECTOR[UPB npv-1]VECI temp;
 altered := TRUE;
 temp[:place-1]:= npv[:place-1];
 temp[place:]:=npv[place+1:];
 npv:=temp
)
```

4. *loc_disp* is the local displayer to be used as a parameter to *inner_read*. It must display any specified rectangle of the picture. The displayer of the picture displays arbitrary parts of

the picture and its parameters are a display frame, an integer and the data of the picture to be displayed. The data of the picture is the value stored in *npv*, the integer is the magnification which, since the display is on the screen, is 1.The display frame is obtained from the picture frame and the rectangle parameter.

```
PROC loc_disp=(RECT r)VOID:
  displayer(DISPFRAME pf RESTRICT r,1,npv);
```

5. *redisplay* redisplays the picture. It does this by determining the rectangle of the picture actually on the screen, using the operator *VISIBLERECT* of mode *(PICTUREFRAME)RECT*. It then clears this area using the operator *CLEAR* of mode *OP(PICTUREFRAME,RECT)VOID* and displays the picture in this area using *loc_disp*.

```
PROC redisplay = VOID:
 (RECT pr = VISIBLERECT pf;
  pf CLEAR pr; loc_disp(pr)
 );
```

The editor uses these procedures in an obvious manner to implement the actions of the picture editor. There are tests to ensure that attempts to delete and move when there are no preferred points fail.

§ 8.3 Transferring pictures to other machines

The procedures defining the operations of a picture exist on a filestore of a machine. If a picture is to be transferred to another machine, then it has to access the defining procedures of the operations. Use of the procedures on the machine from which the picture was transferred is unsatisfactory because it would require communication between the two machines. At best, this would be slow, since the displaying of a picture on a screen requires the passage of much information; at worst, there may not be a link between the two machines. To transfer a picture from one machine to another, the procedures defining the operations of a picture must exist on both machines. The transference of a picture will therefore involve the transference of the data of the picture plus a means of identifying the defining procedures of the picture class on the receiving machine. The identification of the picture is by means of a name, which must consist of less than 26 characters.

§ 8.3.1 The naming of the operations of a PictureDefinition

The procedures defining the operations of a picture class are named by naming, either permanently or temporarily the Picture Definition by which they are defined.

When transferring a picture from one machine to another it is only necessary for the procedures to be named on the receiving machine.

§ 8.3.2 The passing of values between machines

To pass a value from one machine to another requires it to be expressed in terms of universal objects such as characters or integers. In the Flex operating system there exist procedures to express general graphical blocks in terms of these universal objects and, in consequence, to transfer a picture between machines its data must be coded as a sequence of characters, integers and graphical blocks.

The characters, integers and graphical blocks are stored in three flexible vectors and these vectors are passed between machines. There are two operations required of a picture definition to transfer its data from one machine to another; one to store the data in the vectors, and one to remove the data from the vectors. These two procedures are called "ed_out" and "ed_in" respectively. If these operations are not present, then it will not be possible to pass pictures from one machine to another.

§ 8.3.3 The ed_out

The Algol68 mode of the procedure defining the ed_out operation is

```
PROC(PROC(CHAR)VOID,
     PROC(INT)VOID,
     PROC(GBLOCK)VOID,
     PICVAL
    )VECTOR[]CHAR
```

The three procedural parameters are the procedures to add values onto the end of the vectors, the *PICVAL* parameter is the data of the picture. The *VECTOR[]CHAR* delivered is the name by which the procedures defining the operations of the pictures on the target machine can be identified.

The example of the ed_out for the polynomial_fit is

```
PROC ed_out = (PROC(CHAR)VOID addc,
                PROC(INT)VOID addi,
                PROC(GBLOCK)VOID add_gb,
                PICVAL pv
               )VECTOR[]CHAR:
(addi(UPB pv);
 FORALL pv_i IN pv DO addi(xOFpv_i);addi(yOFpv_i)OD;
 "polynomial_fit"
);
```

It only uses the integer vector and adds the upper bound of the vector of *VECI*, followed by the x and y components of each of the *VECI*. Finally the name of the picture definition on the target machine is delivered.

§ 8.3.4 The ed_in

The Algol68 mode of the procedure defining the ed_in operation is

PROC(PROC CHAR, PROC INT, PROC GBLOCK)PICVAL

The three procedural parameters are the procedures to remove values from the vectors, the values are removed in the order in which they were added; the *PICVAL* result is the data of the picture created from the values in the three vectors.

The ed_in procedure for the polynomial fit is

```
PROC ed_in = (PROC CHAR nextc,
              PROC INT nexti,
              PROC GBLOCK next_gb
             )PICVAL:
(PICVAL pv = HEAP VECTOR[nexti]VECI;
 FORALL pv_i IN pv DO pv_i:=(nexti,nexti) OD;
 pv
)
```

This procedure reads the upper bound of the vector of *VECI* from the integer vector, using it to generate a vector of the correct size. It then assigns to each element of the vector in turn the next two elements from the integer vector.

§ 8.4 Transferring pictures to and from disc

If a document containing a picture is to be put onto backing store, then operations capable of placing the data of the picture on backing

store and retrieving the data of the picture from backing store have to be defined.

It is possible to store and retrieve integers and characters on backing store and the *ed_out* and *ed_in* operations of the picture provide a coding and decoding of the data of a picture into characters and integers. If the picture definition has no other operations defined for storing and retrieving the data of the picture on backing store, and it has *ed_in* and *ed_out* defined, when a picture is put onto backing store it uses *ed_in* and when it is retrieved from backing store it uses *ed_out*.

It may be that some pictures contain objects that cannot be coded into integers and characters but can be stored on disc, in which case the ed_out and ed_in procedures would not be suitable for coding the data. To overcome this problem, it is possible to define two other operations, *val_to_disc* and *disc_to_val* for transferring the data of a picture to and from backing store. The decision as to which operations are used by the Flex operating system when a document containing a picture is put onto backing store is as follows. If *val_to_disc* is defined then it is used, otherwise *ed_out* is used; if neither of these is present, then the document cannot be stored on backing store. If the data of a picture is stored using *val_to_disc*, then it is retrieved using *disc_to_val*; if it is stored using *ed_out*, it is retrieved using *ed_in*. If the necessary retrieval procedure is not defined for the picture, then the picture can not be retrieved from disc.

§ 8.4.1 The val_to_disc

The Algol68 mode of the procedure defining the *val_to_disc* operation is

PROC(PICVAL)INT

Its parameter is the data of the picture and it delivers a disc capability of its disc representation. In Algol68 the disc capabilities are represented by values of mode *INT*.

In the case of the polynomial fit the procedure for transferring the *PICVAL* to disc is simple. The *PICVAL* consists of a vector of pairs of integers and *pb_to_d* (kept in pb_to_d :Module) accepts this as a parameter and delivers a disc_capability of it.

```
PROC val_to_disc = (PICVAL pv)INT:pb_to_d(pv);
```

§ 8.4.2 The disc_to_val

The Algol68 mode of the procedure defining the disc_to_val operation is

PROC(INT)PICVAL

This takes the disc capability of the disc representation of data of the picture and should deliver the data of the picture.

The *disc_to_val* of the polynomial fit picture is again simple, requiring a call of *from_disc* (kept in from_disc :Module).

```
PROC disc_to_val = (INT d_ptr)PICVAL :from_disc(d_ptr,( ))
```

§ 8.5 Splitting pictures

If a document is being prepared for printing, it may be necessary to split it into pages. There are procedures in the Flex operating system to process a document into pages of some fixed length. The boundaries between pages cannot be determined arbitrarily since they may lie in unreasonable places, for example in the middle of a line of characters.

```
This is not quite on the bottom of the page
```

A line of characters cannot be split. This is not true of a vertical composition of graphical blocks, which can be split at any boundary between the graphical blocks from which it is formed. The decision of where to split a graphical block is dependent on the type of graphical block.

To determine where to split a picture there is an operation v_split defined by an Algol68 procedure with mode

```
PROC(PICVAL pv,
     VECI size,
     INT ypt,
     INT place,
     PROC(PICVAL,VECI,INT)GBLOCK maker
    )STRUCT(GBLOCK a,b)
```

pv is the data of the picture, *size* is the size of the picture and *ypt* its ypt. *place* is the displacement from the top of the picture, where the calling procedure would like the picture to be split. *maker* is a procedure for making a graphical block of the picture from elements of the data class of the picture (c.f. section "The making of a picture from a picture definition"). The result of the procedure must be two graphical blocks, the first of which has a height less than or

equal to *place*. These two graphical blocks represent the splitting of the picture and have the same width as the picture. If the picture cannot be split, then the first block should be a graphical block of size *(0,*width of picture*)* and the second the graphical block of the entire picture.

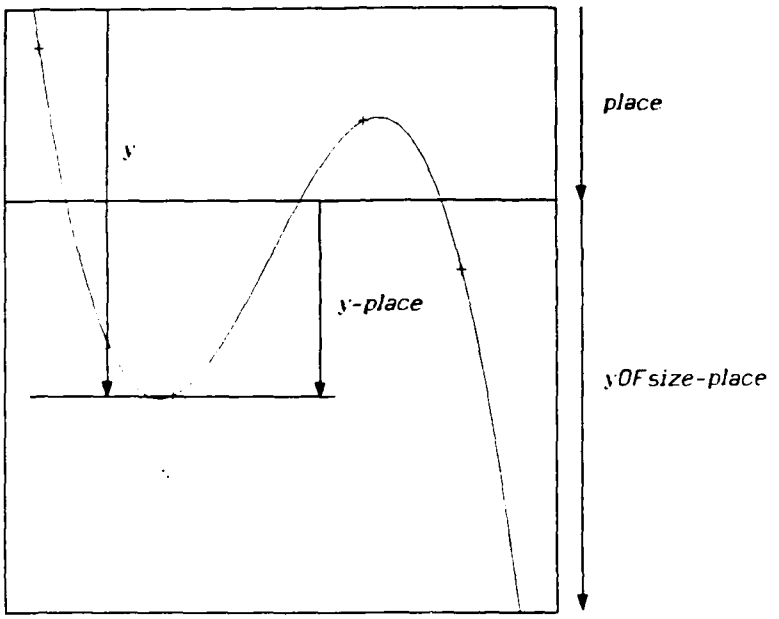
If a v_split procedure is not included in a Picture Definition, then the picture will not be split.

Following is the v_split procedure for polynomial fit. This is artificial as it would not normally make sense to view a graph in two parts

```
MODE PICVAL = REF VECTOR[]VECI;
MODE VSRES = STRUCT(GBLOCK a,b);

PROC v_split = (PICVAL pv,
                VECI size,
                INT ypt,
                INT place,
                PROC(PICVAL, VECI,INT)GBLOCK maker
               )VSRES:
(maker(pv,(xOFsize,place),ypt),
 (REF VECTOR[]VECI npv = HEAP VECTOR[UPB pv]VECI;
  FORALL npv_i IN npv, pv_i IN pv DO
   npv_i := (xOFpv_i, yOFpv_i - place)
  OD;
  maker(npv,(xOFsize,yOFsize-place),0)
 )
)
```

This procedure decides to split the picture at the point indicated by *place*. It does this by returning for the first graphical block the same picture with a smaller vertical size; it can do this using the procedure *maker*. The second graphical block is again the same picture, only this time the y coordinates of all the points have been decreased by *place*, as has the y coordinate of the size of the picture. The second graphical block is created from a new vector with these new points.



§ 8.6 Using pictures as input to compilers

Documents on Flex can be used as input to compilers. If such a document contains a picture, then it may be necessary for the compiler to interpret the picture. Compilers can not be expected to interpret all possible pictures. At the time of creation the compiler does not know all the pictures which it may have to interpret.

There is an operation, called *make_gcomp_input* which can be defined as part of a Picture Definition, which provides an interpretation of the picture for compilers. If this operation is not included in the definition of a picture, the picture is ignored by the compiler.

Compilers also need to be able to indicate where on a document errors occur. The Flex operating system does this by means of marks. Marks cannot be stored on disc, so that when a document containing marks is to be stored, the document is stored on disc as usual, but the marks are collected together to form a data structure of Algol68 mode *INDEX*, kept in index_modes_ :Module. When the document is retrieved from disc the marks can be folded back onto the document.

§ 8.6.1 Marks and indices

A mark is a vector of integers which specify a location in a document.

When a compiler discovers an error in the text of a program, it has to specify where it is and the message that it wishes the user of the compiler to see. During a complete compilation it may find more than one error and an index is the collection of all the marks together with their error messages.

§ 8.6.2 make_gcomp_input

make_gcomp_input is an operation of a picture defined by a procedure with Algol68 mode

```
PROC(PICVAL)INPUTRES
```

INPUTRES is the Algol68 mode

```
STRUCT(PROC UNION(RES,VOID) reader,
       PROC REF VECTOR[]INT current
      )
```

This procedure is applied to the data of the picture and delivers two procedures *reader* and *current*. *reader* is responsible for providing understandable units to the compiler and *current* is responsible for marking the picture to enable the compiler to mark errors.

§ 8.6.2.1 The reader

The *reader* is a procedure of mode

```
PROC UNION(RES,VOID)
```

It is assumed that compilers read documents from start to finish reading one unit of data at a time. The reader is a procedure which when called provides the next unit of data understandable to the compiler. If the *VOID* union is delivered then the reader has reached the end of the picture. *RES* is the mode of the data acceptable to the compiler.

RES is a *UNION(DECLINE, GBLOCK)*
and
DECLINE is a *STRUCT(LINE line, FONT font, BITS mode, forbid)*

DECLINE is essentially a line of text with some information about the manner in which it was constructed, for example if a line was superscripted this information would be passed in *mode*. *RES* is a union of the line and a graphical block

§ 8.6.2.2 The current

The *current* is a procedure which when called delivers the mark of the current position of the reader.

§ 8.6.3 The folding of indices

Indices are represented by the Algol68 mode *INDEX* where

```
INDEX = REF VECTOR [] INDEXITEM,
INDEXITEM = STRUCT(INT place, INDEXTL tl),
INDEXTL = UNION(LINE,INDEX)
```

This mode is a little different from the *REF VECTOR []INT* of the mark delivered by *current* of *make_gcomp_input*. An *INDEX* is essentially a set of marks with a message attached to each. In an index, each mark is represented by a list of integers, *place*, instead of the vector provided by *make_gcomp_input* and is terminated by a message. This should be thought of as a many branch tree with the leaves being the messages. There is generally one more integer in the list than there was in the mark. This is because the unit delivered by *make_gcomp_input* can be a line and the index refers to a point within that line.

To mark a picture, it is necessary to fold into the picture all the marks specified by an index. This requires the specification of an operator in the definition of the picture called *fold_index* which takes the data of the picture and the index, delivering the data corresponding to the marked picture.

The Algol68 mode of the procedure defining *fold_index* is

```
PROC(PICVAL,INDEX)PICVAL
```

If a picture is to be marked, then it must have the ability to store the marks as part of its data.

Marks are not stored on backing store. When a picture with marks is put onto backing store, it is necessary to store the unmarked version of the data of the picture, delivering an index to the marks in addition to the disc capability of the coding of the unmarked data. When the data of the picture is retrieved from backing store, the marks can be added to the data using *fold_index*. To store the data of the picture separately from its marks requires the definition of an operator in the picture definition called *to_disc*, to do this.

The Algol68 mode of the procedure defining *to_disc* is

PROC(PICVAL)GBDRES

where *GBDRES = STRUCT(INT disc, INDEX index)*

disc is the disc capability of the coding of the data of the picture, and *index* is the index of marks of the picture.

§ 9 Internal graphical blocks

In the cases of vertical and horizontal composition of graphical blocks, the operations of the composite block are expressed in terms of the internal blocks. For example the displayer of a horizontal composition is dependent on the displayers of each of the component blocks.

It is possible for pictures to be recursively defined in the same manner, that is pictures can be composed from other graphical blocks, and there is a means of using the operations of an internal block from with an operation of the picture.

For each operation in the abstract structure of a graphical block there is a procedure, available to the procedures defining the picture operations, to apply the operation to the block. This section describes how to apply these operations to the internal blocks.

§ 9.1 The internal displayer

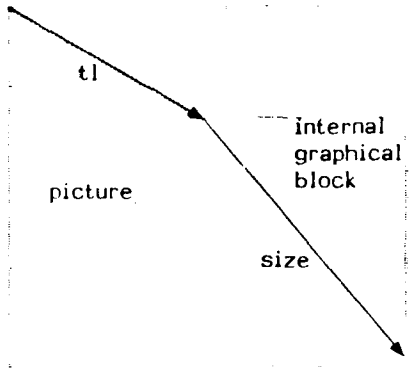
The procedure for displaying internal graphical blocks is *disp_gb* and is kept in the module disp_gb_m :Module. It is a value of mode

PROC(DISPLAYFRAME,INT,GBLOCK,FONT,BITS)VOID

It is a value of similar mode to the displayer of the picture. *FONT* and *BITS* are historical and unused by the procedure, *GBLOCK* is the internal graphical block, *INT* is the magnification at which the block is to be displayed, and *DISPLAYFRAME* is its display frame.

The display frame of graphical block is the structure holding information on where to display the block. The display frame of an internal block depends on the display frame of the picture and where the internal block is within the picture.

To determine the display frame of the internal graphical block, there is a procedure called *inner_frame* kept in display_fram :Module of mode *PROC(DISPLAYFRAME,RECT)DISPLAYFRAME*.



In this example the display frame of the internal block is determined by the display frame of the picture and the rectangle *(tl,size)*.

§ 9.2 The internal editor

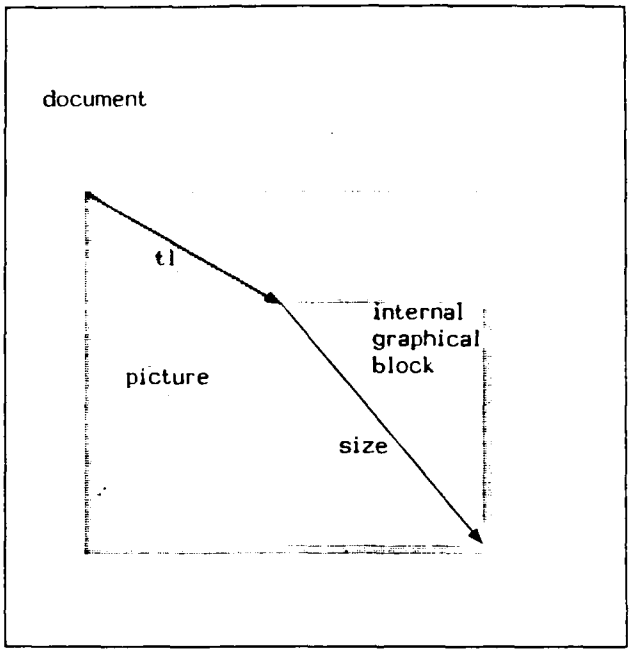
The procedure for editing internal graphical blocks is *edit_gb* and is kept in the module pic_edit_gb_ :Module. It is a value of Algol68 mode

PROC(PICTUREFRAME, INT, GBLOCK)GBEDITRES

where
GBEDITRES = STRUCT(BOOL altered, INT reason, GBLOCK gblock)

It is the direct parallel of the editor of the picture only it edits a graphical block instead of the data of a picture.

The major problem of using the editor of an internal graphical block is in the provision of the *PICTUREFRAME* parameter. The picture frame of the picture holds information on the context of the picture, i.e. information on what is outside of the picture. The same must be true of the picture frame of the internal block editor, and the picture frame of the internal block can be calculated from the picture frame of the picture plus information about the area between the boundaries of the picture and the internal graphical block.



The information that must be provided to create a pictureframe for the internal block is where it is, provided by means of a rectangle (*tl*, *size*), how to display the shaded area and what the picture should do if the internal graphical block should request an expansion, provided by means of two procedures.

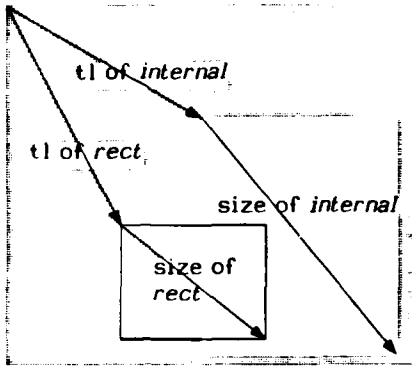
§ 9.2.1 The display procedure in the creation of a PictureFrame

The display procedure required in the creation of a picture frame is responsible for displaying the picture in the area outside of the internal block.

As the internal block is edited, it may expand or contract and change its form, making the area to be displayed different at different calls of the displayer. Since the displayer does not have to display the internal block, it does not need to know what changes are made to the internal block, it only needs to know where the internal block is. This is passed to the displayer as a parameter.

The Algol68 mode of the display procedure is

PROC(RECT rect, RECT internal)VOID



The display procedure should display everything that is in *rect* but not in *internal*.

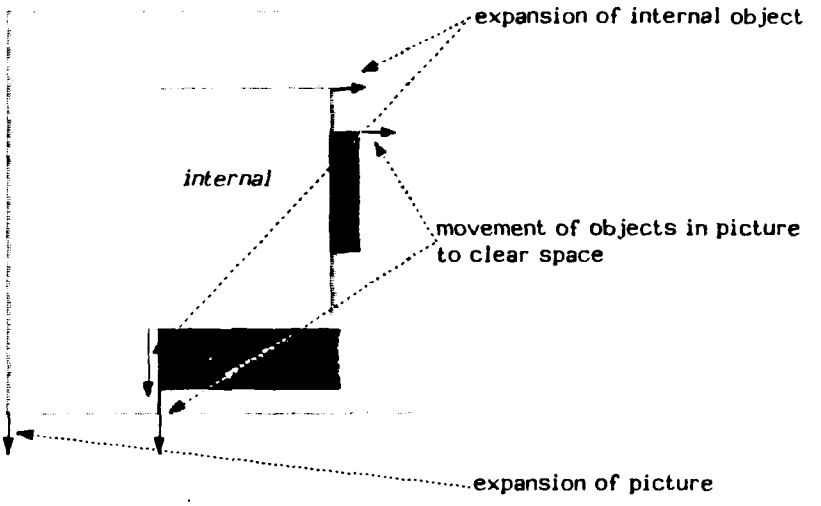
§ 9.2.2 The expand procedure in the creation of a PictureFrame

The expand procedure has an Algol68 mode of

PROC(RECT internal, VECI amount)BOOL

internal is the rectangle occupied by the internal graphical block, *amount* is the size of the expansion requested by that block and can either be positive or negative depending on whether an expansion or contraction is requested. The purpose of the procedure is firstly to determine whether or not the requested expansion can be allowed and secondly to make any changes to the picture that are necessary to accommodate the expansion. The *BOOL* result is whether or not the expansion is allowed. A contraction is always allowed.

If an expansion is allowed, it is a requirement of the expand procedure that it clears an area for the internal block to expand into. The clearing of this area may make it neccessary to increase the size of the picture.



If the picture has to expand it will have to expand using the operator *EXPAND* as described in the section on the editor. If the picture can accommodate the increase in size of the internal block it should clear the space required by the internal block and deliver *TRUE*, otherwise it should do nothing and deliver *FALSE*.

If the internal graphical block wishes to contract, it will have already cleared the space from which it has contracted. The expander should then move any objects it desires into the resulting space and if this results in a decrease in size of the picture, it should call the operator *EXPAND* to reduce its size.

There is a restriction on the expand procedure. A contraction of the internal block cannot produce an expansion of the picture. This restriction is forced because the internal block assumes that its contraction will be allowed and an expansion of the picture might not be allowed.

The procedure for creating the picture frame for the internal graphical block is *new_frame* kept in picture_fram :Module. Its Algol68 mode is

```
PROC(RECT place,
     PROC(RECT,VECI)BOOL expander,
     PROC(RECT,RECT)VOID displayer,
     PICTUREFRAME pf
    )PICTUREFRAME
```

place is the rectangle of the internal block, the *expander* is the

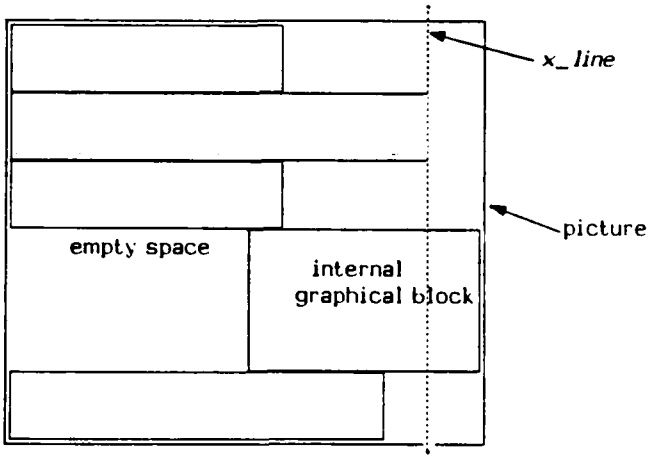
expand procedure, *displayer* is the display procedure and *pf* is the picture frame of the picture. The picture frame of the internal block is delivered.

§ 9.2.3 Regular PictureFrames

In many cases, the required picture frame of an internal graphical block takes one of three forms. To cater for these three common cases there are three procedures to create the picture frames. All the procedures require a displayer and this should be of the Algol68 mode *PROC(DISPLAYFRAME)VOID* and should consist of the displayer of the picture with the data of the picture and the magnification bound to it.

The problems of creating new picture frames rest solely with expansion; if there were no expansion then there would be no expander and the display procedure could be deduced from the displayer of the picture. The three procedures to create picture frames reflect three different methods of expansion and are kept in the module x_and_y_fram :Module.

The first form of expansion is one that preserves horizontal lines.



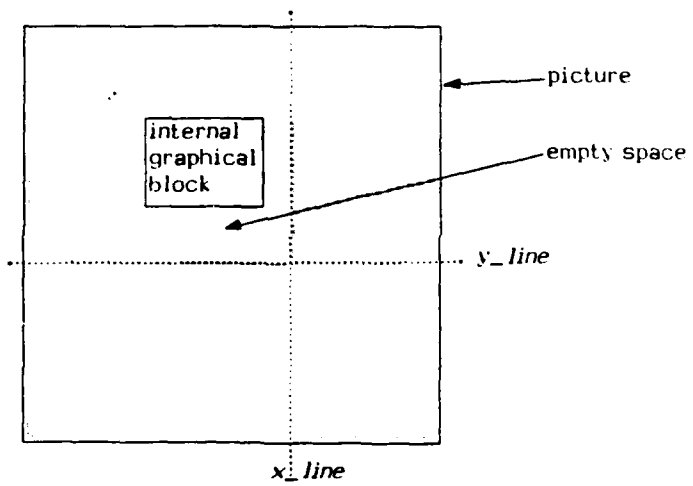
If the internal graphical block expands in the y-direction, then everything below it is moved in that direction by the amount of the expansion and the picture has to expand by the same amount. If there is an expansion in the x-direction, then the internal block expands freely until it gets to the *x_line*. The *x_line* is the right most line of the picture, not including the internal block; expansions of the internal block beyond the *x_line* cause corresponding expansions in

the picture. The procedure to create this form of pictureframe is *y_new_frame* and is of mode

```
PROC(RECT internal,
     PROC(DISPLAYFRAME)VOID displayer,
     INT x_line,
     PICTUREFRAME pf
    )PICTUREFRAME
```

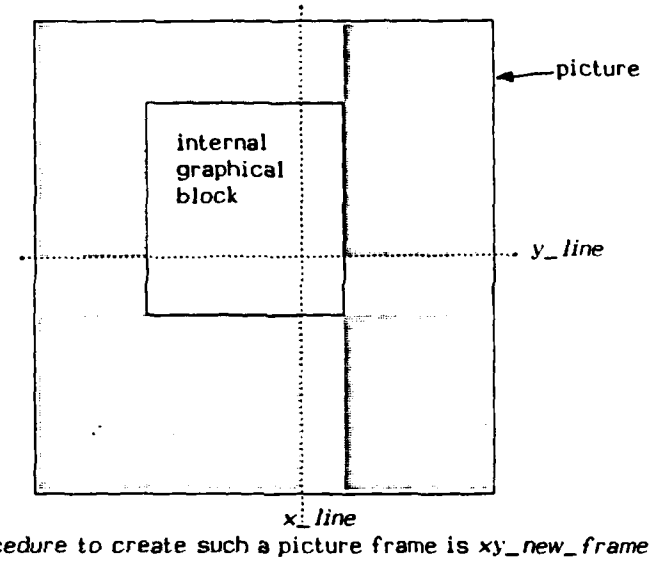
There is a procedure of identical mode for creating pictureframes preserving vertical lines defined in a similar manner called *x_new_frame*.

The third form of expansion preserves horizontal and vertical lines.



The *x_line* and *y_line* must be beyond the internal graphical block and there must be empty space between the block and these lines. The internal block is allowed to,expand freely until it meets either the *x_line* or the *y_line*, in which case everything beyond them is shifted to the right or the left causing an expansion of the picture.

This is illustrated in the following diagram.



The procedure to create such a picture frame is *xy_new_frame* and is of mode

```
PROC(RECT internal,
     PROC(DISPLAYFRAME)VOID displayer,
     INT x_line, y_line,
     PICTUREFRAME pf
    )PICTUREFRAME
```

§ 9.3 Transferring pictures to other machines

To transfer an internal graphical block between machines involves placing it in the three vectors of *INT*, *CHAR* and *GBLOCK*. Since an internal graphical block is represented by a *GBLOCK*, it can be stored in the *GBLOCK* vector.

§ 9.4 Transferring pictures to and from disc

There is a procedure *gb_to_disc*, kept in gb_to_disc_m :Module, with Algol68 mode *PROC(GBLOCK, VECI)GBDRES*, where *GBDRES* is a *STRUCT(INT disc, INDEX index)*. The *VECI* parameter is irrelevant. The procedure takes a graphical block and delivers a disc capability for its backing store coding and an index. For further details on the index see the section on using pictures as input to compilers.

To retrieve a graphical block from disc, there is a procedure

disc_to_gb, (kept in disc_to_gb_m :Module) of mode *PROC(INT,FONT)DTGRES*.
The *INT* parameter is the disc capability of the backing store coding of the graphical block, and the *FONT* parameter is irrelevant. *DTGRES* is a *STRUCT(GBLOCK gb,VECI tlcursor)*; *gb* is the graphical block retrieved from disc and *tlcursor* can be ignored.

§ 9.5 Splitting pictures

To split graphical blocks internal to a picture, there is a procedure *vsplit*, kept in vsplit :Module, of Algol68 mode

PROC(GBLOCK gb, INT place, FONT ft)STRUCT(GBLOCK a,b)

This procedure attempts to splits *gb* into two graphical blocks *a* and *b* at the point *place*. *ft* is not used.

§ 9.6 Using pictures as input to compilers

To provide the compiler input of a graphical block there is a procedure *make_input* of mode *PROC(GBLOCK)INPUTRES*, kept in pic_make_inp :Module, which delivers two procedures, one for providing units understandable to the compiler and the other for marking the internal block.

Similarly there is a procedure *fold*, kept in fold_m :Module, of mode *PROC(GBLOCK,INDEX,FONT)GBLOCK* which folds the index parameter into the internal block. The font parameter is unused.

To store a marked graphical block on disc, the procedure *gb_to_disc*, as described in the section on transfer to and from disc, is used. The index delivered as part of the result of *gb_to_disc* is an index to the set of marks in the internal block.

§ 10 A simple structure editor

To illustrate some of the more complex operations that can be applied to pictures, the *displayer* and *editor* of a small structure editor will be discussed in some detail.

The *editor* is intended to be one that manipulates the structure of a simple program language. It is not intended to be realistic, but only to illustrate the points about graphics being made.

The program language consists of sequences of statements, with a statement being either an assignment or a conditional. A conditional consists of three sequences: the condition, the sequence obeyed when the condition is true and the sequence obeyed when the condition is false.

This structure can be represented by the Algol68 mode *SEQ* where

```
SEQ = REF VECTOR []STATEMENT
STATEMENT = UNION(ITEF, ASGMT)
ITEF = STRUCT(SEQ condition, t, f)
ASGMT = STRUCT(LINE lhs, rhs)
```

The size of the graphical representation of any picture is important. In the case of the structure editor this is determined entirely by the data of the picture (cf. the polynomial fit which is not), and its size is determined by the manner in which the data is displayed.

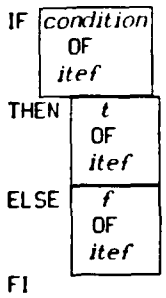
A value *asgmt* of mode *ASGMT* is displayed as the concatenation of the three vectors of characters *lhsOFasgmt*, " := ", and *rhsOFasgmt*, in standard font. Every character in standard font is size *(9,15)*, making the size of *asgmt*

*((UPB lhsOFasgmt + UPB rhsOFasgmt +4)*9, 15)*

A value *seq* of mode *SEQ* is displayed by displaying each of its statements vertically above each other with a left justification. In the case where there are no statements the vector of characters "SKIP", in standard font is displayed. So the procedure for determining the size of a *seq* is

```
IF UPB seq = 0 THEN (36,15)
ELSE
   (maximum of x components of seq,sum of y components of seq)
FI
```

A value *itef* of mode *ITEF* is displayed as follows



making its size
(the maximum of
(27 {size of "IF "} + x-component OF *conditionOFitef*,
45 {size of "THEN "} + x-component OF *tOFitef*,
45 {size of "ELSE "} + x-component OF *fOFitef*),
sum of y components of *condition*,*t* and *f* of *itef*+15 {size of "FI"}).

The sizes of the various structures can always be calculated using the above algorithm, but to avoid recalculation of these sizes, they are stored in the data structure, making the actual mode of the values manipulated by the structure editor

```
GSEQ = STRUCT(VECI size, REF VECTOR []GSTATEMENT seq)
GSTATEMENT = STRUCT(VECI size, UNION(GITEF, GASGMT) stat)
GITEF = STRUCT(VECI size, GSEQ condition, t, f)
GASGMT = STRUCT(VECI size, LINE lhs, rhs)
```

§ 10.1 The displayer of GSEQ

The manner in which a value of mode *GSEQ* (a sequence) is to be displayed is that each statement in the sequence will be displayed one below the other with a left justification. If there are no statements in the sequence then the word "SKIP" will be displayed. No matter what the font of the document in which it is contained, the font of a *GSEQ* will be the standard font. Each character of the standard font is 9 pixels wide and 15 pixels high. This makes the size of "SKIP" 36 x 15.

When a displayer is called, it is not always required to display the complete picture (e.g. in scrolling the screen where usually only a small part of the picture needs to be displayed). For each part of the picture, it is worth calculating whether or not that part of it is required to be displayed. For example, *display_seq*, the displayer of *GSEQ* is written in terms of *display_statement*, the displayer of a *GSTATEMENT*. It would be possible to write the displayer of *GSEQ* by

calling *display_statement* on each of the statements, even though some of the statements may not lie in the area to be displayed. This relies on *display_statement* being able to determine whether or not it has to display the statement. If this attitude is adopted consistently it is only procedures which actually write directly to the screen (*display_seq* does not write to the screen, but calls other procedures to do so) that have to decide whether or not they have an area to write to. In this case it takes almost as long to display a small part of the sequence as it does to display it in its entirety.

For this reason, except for the displaying of "SKIP", if a statement does not lie in the area to be displayed, *display_statement* is not called to display it.

The procedure for displaying *GSEQ* treats "SKIP" as a special case. In the displaying of the sequence, it finds the first statement that intersects the area of the picture which is to be displayed and continues displaying until it is below that area, firstly checking that the statement to be displayed is not entirely to the left of the area to be displayed.

```
PROC display_seq = (DISPLAYFRAME frame, INT mag, GSEQ seq)VOID:
(RECT visible_rect = VISIBLERECT frame;
 REF VECTOR []GSTATEMENT seq = seqOFseq;

 INT top = yOFtlOFvisible_rect,
     bottom = top + yOFsizeOFvisible_rect,
     left = xOFtlOFvisible_rect,
     u = UPB seq;

 IF u = 0 THEN {No Statements}
   GBLOCK gb = flgb("SKIP", standard_ft); {make GBLOCK for "SKIP"}
   RECT skip_rect = ((0,0),(36,15)); {rectangle occupied by "SKIP"}
   disp_gb(inner_frame(frame,skip_rect),mag,gb,standard_ft,16r0)
 ELSE
   INT start := 0,place :=1;
   {place is the number of the current statement
    start is the position of the top of the current statement}

   WHILE
     INT depth = start + y OF (size OF seq[place]);
     place < u AND depth <= top
   DO
     place +:= 1;start := depth
   OD;
   {find the first statement in the visible rectangle}

   WHILE start <= bottom AND place <= u DO {Whilst in visible rect}
     VECI size = size OF seq[place];
     IF xOFsize > left THEN{statement is not entirely to left of area}
       RECT stat_rect = ((0,start),size);
                                  {rectangle occupied by statement}
       display_stat(inner_frame(frame,stat_rect),mag,seq[place])
     FI;
     start+:=y OF (size OF seq[place]);
     place+:=1
   OD
 FI
)
```

§ 10.2 The editor of GSEQ

The general manner in which editors on Flex operate is to read an action code from the puck/keyboard; if the code is a value which they can act upon, they do so, otherwise they pass it back as an action code to the calling editor. The action code continues to pass outwards until it reaches an editor which can act upon it. This editor, having taken the action, recalls the editors of internal graphical blocks until it reaches the editor of a bottom level object.

This has the effect of making the structure of graphical objects transparent to the user.

One of the purposes of the structure editor is to make the structure visible. This is achieved by making the picture a bottom level object and it does not immediately call any further editors. If it is desired to move down the structure of the picture, then this is done explicitly by pressing the *examine* key on the puck. Any other key will cause the editor to be left with the action code set to the value of the key pressed. The editors called by the picture editor can only be left explicitly, by pressing the result button on the puck, and in this manner they indicate the structure of the picture.

Some system procedures mark documents, this is to indicate places of interest in the document such as a compilation error. These marks are found using the tab key. If the editor is called with the reason of *find_next_mark* then it is looking for one of these marks. There is no general procedure for putting marks into pictures, but if desired they can be put into the underlying data structure. This is not done in the picture of *GSEQ* so, when looking for marks, the editor is left immediately. The editor is left with the action of *find_next_mark* informing the calling editor that it has to look for a mark in the remainder of the document.

If the editor is entered with a reason of *find_string* it is searching for a string of characters in the document. The string is not available to the picture editor so it is left immediately without altering anything. The action passed to the calling editor is *find_string* informing it to look for the string in the remainder of the document.

If the editor is entered for any other reason, it puts the cursor on the entire *GSEQ* showing the outermost level of the structure, moves the document to include the cursor in the window and reads the puck/keyboard. This level of the editor only responds to one key, examine, which causes this editor to edit the seq and then read the keyboard/puck again. All other responses are passed backwards to the calling editor for it to act upon.

There is no need to make a new picture frame to pass to the editor of the sequence, since this has the same frame as the picture.

```
MODE EEDITRES = STRUCT(BOOL altered,INT reason, GSEQ picval);

PROC editor=(PICTUREFRAME frame,INT reason,PICVAL gseq)EEDITRES:
 IF reason = find_next_mark OR reason = find_string THEN
  (FALSE,reason,gseq) {leave editor with no change}
 ELSE
  REF VECTOR []GSTATEMENT seq = seqOFgseq;
  REF VECTOR []GSTATEMENT nseq :=
                      HEAP VECTOR[UPB seq]GSTATEMENT := seq;
                      {copy sequence}
  ref_display_seq := display_seq;
  ref_edit_seq := edit_seq;
                      {assignment for recursive calls}

  BOOL altered := FALSE; {stores alteration to sequence}
  INT action;
  PROC display = (RECT r)VOID:
   display_seq(DISPFRAME frame RESTRICT r,1,(SIZE frame,nseq));
      {local displayer}

  WHILE set_cursor_rect(frame,((0,0), SIZE frame));
                      {put cursor on entire sequence}
   action := read(frame,display); {read keyboard}
   IF action = examine THEN
     {If action is examine then edit sequence}
     EDITRES er = edit_seq(frame,(SIZE frame,nseq));
     nseq := seqOFseqOFer; {store result}
     altered := altered OR alteredOFer;TRUE
     {record alteration and repeat}
   ELSE
     FALSE {leave editor}
   FI
  DO SKIP OD;
  (altered, action, (SIZE frame,nseq))
FI
```

§ 10.2.1 edit_seq

edit_seq is the editor of a sequence. It is designed to show the structure of the picture and so, like its calling editor, will not go deeper into the structure unless explicitly told to by pressing the examine key on the puck. Similarly, the editor can only be left by being told to explicitly, by pressing the result button on the puck.

The operations of *edit_seq* are quite simple. On entry the cursor is placed on the first statement of the sequence, from where it has the ability to move up and down through the sequence. Statements can be

inserted, deleted or edited and finally the editor can be left by pressing result.

```
PROC edit_seq = (PICTUREFRAME frame, GSEQ gseq)EDITRES:
(REF VECTOR []GSTATEMENT nseq:=
            HEAP VECTOR [UPB seqOFgseq]GSTATEMENT:=seqOFgseq;
                {copy gseq}
 INT u := UPB nseq;    {number of statements}
 INT statno := u MIN 1; {index of current statement in vector}
 PROC max = INT:Edfile
 {procedure delivering maximum width of statements not including
  the current statement}

 INT pl := 0; {The top of the current statement}
 BOOL altered := FALSE; {records alterations}
 INT action; {the most recent read of the puck/keyboard}

 PROC display = (RECT r)VOID:
    display_seq(DISPFRAME frame RESTRICT r,1,(SIZE frame,nseq));
    {local displayer}

 set_cursor_rect(frame,
                 ((0, 0),(u=0|(36,15)|sizeOFnseq[statno])));
  {set cursor to cover the first statement}
 WHILE
  action := read(frame,display); {read puck/keyboard}
  IF action=examine THEN Edfile;TRUE
  ELIF action=result THEN FALSE {If action is result leave editor}
  ELIF action=move_up THEN Edfile;TRUE
  ELIF action=ins_asgmt_abv OREL action=ins_itef_abv THEN
    Edfile;TRUE
  ELIF action=ins_itef_bel OREL action=ins_asgmt_bel THEN
    Edfile;TRUE
  ELIF action=delete THEN Edfile;TRUE
  ELIF action=move_down THEN Edfile;TRUE
  ELSE beep;TRUE
  FI
 DO SKIP OD;
 (altered,action,(SIZE frame,nseq))
)
```

To illustrate the operations of *edit_seq* the effects of examine and delete will be considered in some detail.

§ 10.2.1.1 The effect of examine

When examine is pressed on the puck, if the sequence is not empty,

then the current statement is edited, the result of the edit is stored and if there has been an alteration to the statement this is recorded by setting *altered* to *TRUE*. Finally the cursor is put on the new statement.

```
IF u = 0 THEN beep ELSE
  PICTUREFRAME new_pic_frame =
    y_new_frame( ((0,pl), sizeOFnseq[statno] ),
                 (DISPLAYFRAME df)VOID:
                   display_seq(df,1,(SIZE frame,nseq)),
                 max,
                 frame
                ),
  SEDITRES er = edit_statement(new_pic_frame, nseq[statno] );
  nseq[statno] := statOFer;
  altered := altered OR alteredOFer;
  set_cursor_rect(frame,((0, pl), SIZE new_pic_frame))
FI
```

§ 10.2.1.2 The effect of delete

There are three cases to consider when deleting a statement from a sequence. When there are no statements in the sequence delete performs no action. When there is only one statement it is deleted and replaced with "SKIP". When there is more than one statement it is deleted. If the deleted statement was not the last in the sequence, the new current statement is the one following the deleted statement, otherwise it is the preceding one.

When a statement is deleted the sequence will have to contract. The deletion of a statement can be viewed as the contraction of that statement to size (0,0) or the size of "SKIP", depending on whether or not it is the only statement in the sequence. For this reason a picture frame is made for the current statement and this is contracted. The rule of clearing the area from which the statement contracts is obeyed and, if there are no statements left in the sequence, "SKIP" is displayed.

The values recorded in *statno*, *u* and *pl* are updated, the new value of the data recorded in *nseq* and *altered* set to *TRUE*.

```
IF u = 0 THEN beep ELSE {If no statements then beep}
  VECI crtsize=size OF nseq[statno]; {size of current statement}
  PICTUREFRAME new_pic_frame =  {picture frame of current stat}
   y_new_frame( ((0,pl), crtsize),
                (DISPLAYFRAME df)VOID:
                  display_seq(df,1,(SIZE frame,nseq)),
                max,
                frame
               );
  fill_frame(DISPFRAME frame RESTRICT RECT((0,pl),crtsize),1);
   {clear rectangle of current statement}
  IF u=1 THEN {special case as delivers "SKIP"}
    new_pic_frame EXPAND VECI(36, 15)-crtsize;
      {contract current statement by its size - size of "SKIP"}
    nseq := seqOFempty_seq; statno := 1; u :=0;
      {assign empty sequence to n_seq, set and u to 0}
    set_cursor_rect(frame,((0,pl), (36,15)));
      {put cursor on "SKIP"}
    altered  := TRUE;   {record alteration}
    display_seq(DISPFRAME frame,1,(SIZE frame,nseq))
       {display SKIP}
  ELSE
    new_pic_frame EXPAND -crtsize;
       {reduce statement size to (0,0)}
    REF VECTOR []GSTATEMENT t = HEAP VECTOR [u - 1]GSTATEMENT;
    t[1:statno - 1] := nseq[1:statno - 1];
      {make new vector of statements}
    IF statno=u THEN
      {If last statement then make current statement the one before}
      pl -:= yOFsizeOFnseq[statno -:= 1]
    ELSE {make current statement the one after}
      t[statno:u - 1] := nseq[statno + 1:u]
    FI;
    nseq := t;u -:= 1; altered := TRUE;
    {assign new sequence to nseq, reduce u by 1 and record
     alteration}
    set_cursor_rect(frame,((0,pl), sizeOFnseq[statno]))
    {put the cursor on the new current statement}
  FI
FI
```

§ 11 Declaring Flex Picture Procedures

There is a procedure called dec_picture_fns, for use with pictures, which is kept in the common dictionary on Flex. Its curt mode is Filed(Dictionary -> Void). It names and keeps further procedures for use with pictures. They are kept in the dictionary supplied as a parameter to the procedure and any PictureDefinitions created using them will also reside in that dictionary.

new_picture_defn	for making a new PictureDefinition.
amend_picture_defn	for changing a PictureDefinition
make_picture_maker	which takes a PictureDefinition delivering a Compiledpair keeping the procedure for making pictures.

There are procedures for use in Control A whilst the cursor is on a picture:

recover_edfile	which delivering the Edfile of the PictureDefinition from which the picture was created.
recover_picture_defn	which delivering the PictureDefinition from which the picture was created.
return_value	which delivers the data structure of the picture.

§ 12 Conclusion

The Flex implementation of documents described in this paper has been used by various groups at RSRE. At present pictures are being created to draw circuit diagrams corresponding to ELLA descriptions of hardware [J. D. Morison, N. E. Peeling and T. L. Thorp, 1985] to draw MASCOT diagrams [MASCOT Official Handbook, 1987] and to draw Z specification schema.

In addition to these specific uses of pictures, there are more general pictures to augment the Flex editor. These include pictures for drawing graphs and pictures for drawing diagrams.

The uses that have already been made of pictures illustrate that pictures can be used both to give a visual representation to already existing data, as in the case of the ELLA circuit diagrams; and also to enhance the general editing facilities of the existing editor.

There will be an implementation of pictures, defined in terms of Ten15 [P. W. Core and J. M. Foster, 1986]. Ten15 can express procedure values, enabling a similar implementation to that on Flex. Instead of relying on Algol68 types to express and determine the structure of a graphical object, this will be done in terms of the universal Ten15 types. To minimise the inconvenience to users who wish to transport pictures from a Flex system to a Ten15 system, the Ten15 types will be closely related to the Algol68 types.

Acknowledgment

This paper would not be complete without an acknowledgment of the contribution made by Dr. J. M. Foster towards this work. He designed and implemented the basic document and it was his conception to allow users to extend this basic document by identifying the structure of objects on documents.

References

P. W. Core and J. M. Foster
"Ten15: An Overview"
RSRE Memorandum No. 3977, September 1986.

I. F. Currie and J. M. Foster
"Curt: The Command Interpreter Language for Flex"
RSRE Memorandum No. 3522, September 1982.

I. F. Currie, J. M. Foster and P. W. Edwards
Kernel and System Procedures in Flex
RSRE Memorandum No. 3626, August 1983.

J. D. Morison, N. E. Peeling and T. L. Thorp
"The Design Rationale of ELLA, A Hardware Design and Description Language"
Proc Computer Hardware Description Languages and their Applications, pp 303 - 328, Tokyo Japan, 1985.

I. F. Currie, J. M. Foster and P. W. Edwards
"PerqFlex Firmware"
RSRE Report No. 85015, December 1985.

M. Stanley
"The Use of Values without Names in a Programming Support Environment"
RSRE Memorandum No. 3981, November 1985.

M. Stanley
"Using True Procedure Values in a Programming Support Environment"
RSRE Memorandum No 3916, February 1986.

"Official Handbook of MASCOT, version 3.1"
JIMCOM, RSRE, 1987.

Appendix

System Modules for use with Pictures

coord_and_re :Module Keeps *VECI* and *RECT* the Algol68 mode for the representation of displacement vectors and rectangles.

cursor_contr :Module Keeps *give_cursor_rect* and *set_cursor_rect* for reading and setting the cursor position.

disc_to_gb_m :Module Keeps *disc_to_gb* and *DTGRES* for bringing internal graphical blocks off disc.

display_fram :Module Keeps *inner_frame* the procedure for making internal display frames.

disp_gb_m :Module Keeps *disp_gb* the procedure for displaying internal graphical blocks.

fold_m :Module Keeps *fold* the procedure for folding an index into an internal graphical block.

gblock_modes :Module Keeps the mode *GBLOCK*, the mode of the Algol68 representation of graphical blocks.

gb_to_disc_m :Module Keeps *gb_to_disc* and *GBDRES*, for putting internal graphical blocks on disc.

inner_read_m :Module Keeps *inner_read* a procedure for reading the puck/keyboard.

line_styles :Module Keeps *solid*, *white*, *dot*, *dash*, *dash_dot*, *long_dash* which are the predefined line styles.

make_display :Module Keeps *make_display_frame* which is a

PROC(VECI size, INT mag)DISPLAYFRAME

which creates a *REF[,]BOOL* of size *size*mag* for displaying graphical blocks at a magnification of *mag*, then delivering a display frame for the area.

n_displaced_ :Module Keeps operators +, -, and *DISPLACED* for operating on displacement vectors and rectangles.

pic_draw_m :Module Keeps *pic_draw_line*, *draw_horiz*, *draw_vert* and *fill_frame* for drawing lines and filling in rectangles.

pic_edit_gb_ :Module Keeps *edit_gb* the procedure for editing internal graphical blocks.

pic_make_inp :Module Keeps *make_input* and *INPUTRES* for making compiler input from internal graphical blocks.

pic_move_m :Module Keeps *move* which is a

```
PROC(PICTUREFRAME pf, RECT r, VECI amount)VOID
```

which, within the picture frame *pf* moves the display in the rectangle *r* by the displacement *amount*.

pic_new_wind :Module Keeps *new_window* which is a

```
PROC(PICTUREFRAME pf,
     RECT r,
     GBLOCK gb,
     INT reason,
     PROC(RECT)VOID local_displayer
    )WINRES
```

where

```
WINRES = STRUCT(BOOL altered,
                INT reason,
                GBLOCK gblock,
                VECI tl_cursor,
               )
```

Within the picture frame *pf*, a window is made *r*, in which is edited *gb* with reason *reason*. The *local_displayer* is the displayer of an arbitrary rectangle within the picture frame. The resulting graphical block is delivered, together with reason of leaving, whether it has been altered and the cursor position when it was left.

pic_operator :Module Keeps *DISPLAYFRAME* and *PICTUREFRAME* and the following operators.

SIZE	operates on a display frame or a picture frame delivering the size of the picture they are framing.
AREA	operates on a display frame or a picture frame, delivering a *REF [,]BOOL* which is the area that the picture occupies.
RESTRICT	operates on a displayframe and a rectangle, restricting the area of the displayframe to the rectangle.

DISPFRAME operates on a picture frame, delivering the display frame corresponding to the picture frame.

CLEAR operates on a picture frame or a display frame and a rectangle, clearing that area of the picture specified by the rectangle.

EXPAND operates on a picture frame and a vector displacement, requesting an expansion of the picture frame and delivering a boolean result depending on whether the expansion was allowed.

VISIBLERECT operates on a picture frame or a display frame and delivers that rectangle of the picture to which it is possible to write(i.e. in the case of a picture frame that part of the picture in the window)

pic_reasons_ :Module Keeps names for the reason and action codes.

pic_scroll_m :Module Keeps *pic_scroll* for scrolling the document containing a picture behind the window.

picture_proc :Module Keeps *find_proc* and *find_val* for obtaining the defining procedures of a picture and the data of a picture.

read_m :Module Keeps *read* a procedure for reading the puck/keyboard.

read_puck_m :Module Keeps *read_puck* and *PUCKRES*, for reading the puck.

x_and_y_fram :Module Keeps *y_new_frame*, *x_new_frame* and *xy_new_frame* for making new picture frames.

DOCUMENT CONTROL SHEET

Overall security classification of sheet .....UNLIMITED.....

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification eg (R) (C) or (S))

1. DRIC Reference (if known)	2. Originator's Reference	3. Agency Reference	4. Report Security Classification
	R87011		UNLIMITED

5. Originator's Code (if known)	6. Originator (Corporate Author) Name and Location
778400	RSRE, St Andrews Road, Malvern, Worcs. WR14 3PS

5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location

7. Title
USER-EXTENSIBLE GRAPHICS USING ABSTRACT STRUCTURE

7a. Title in Foreign Language (in the case of translations)

7b. Presented at (for conference papers) Title, place and date of conference

8. Author 1 Surname, initials	9(a) Author 2	9(b) Authors 3,4...	10. Date	pp. ref.
CORE, P.W.			1987.08	75

11. Contract Number	12. Period	13. Project	14. Other Reference

15. Distribution statement

Descriptors (or keywords)

continue on separate piece of paper

Abstract

A means of creating an editor which allows its users to extend the classes of objects manipulated by it is described. This has been achieved by creating an abstract structure representing object classes. An example of such an editor has been implemented on Perq Flex making use of true procedure values.

END

DATE
FILMED

6 88